Lattice

SAS/C® Compiler for AmigaDOS™

User's Manual

Version 5.10

The C Development System for the Amiga

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C® Compiler for AmigaDOS™

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Preface

The Lattice Amiga C Compiler is a complete development system for the Commodore Amiga. This is a reference manual for the Lattice Amiga C Compiler. Its primary purpose is to accurately describe all commands, utility programs, editor, debugger, environment variables, functions and external names that you will normally use while writing and executing C programs.

Contents

This manual consists of seven major sections:

User's Guide (G) This section describes the installation and hardware requirements for the Lattice Compiler. It describes the directories and files and logical name assignments. It describes the Lattice assembler, and presents information on AmigaDOS as it impacts the compiler. It covers compiler and linker operation,and provides a Lattice C language definition. It contains a list of compiler and linker error messages. This section has the pagecode "G" next to each page number.

Utilities (U) This section describes the new programming utilities included with the compiler. These include a profiler,

a traceback facility, a make facility, build and extract tools, global search and replacement tools, a C cross-reference facility, file manipulation utilities, and a word count utility. This section has the pagecode "U" next to each page number.

Commands (C)
This section summarizes all of the commands in the Lattice Amiga C Compiler, including compiler, linker, assembler, and utilities. This section has the pagecode "C" next to each page number.

Editor (E)
This section describes the full screen Lattice Screen Editor (LSE) which is custom tailored for the Amiga. It describes how to use the editor and how it is integrated into the compiler for in-memory edit and compile sessions. This section has the pagecode "E" next to each page number.

Debugger (D)
This section describes the Lattice full-screen symbolic debugger (**CodePRobe**) which is custom tailored for the Amiga environment. It describes how to use the debugger and how to take advantage of tis advanced features, such as debugging multi-tasking processes. This section has the pagecode "D" next to each page number.

Library Reference (L)
This section describes all of the library functions in the Lattice AmigaDOS C Library. This section has the pagecode "L" next to each page number.

Master Index (I)
Each section has its own index and table of contents. The master index provides one section that indexes the entire product. This section has the pagecode "I" next to each page number.

New Features

The Lattice Amiga C Compiler Version 5.0 is a upgrade to the Lattice Amiga C Compiler Version 4.0. The major new features for Version 5.0 include:

680x0/6888x The version 5.0 contains code generation support for the entire family of processor chips including 68000, 68010, 68020, and 68030 processors, and the math co-processor chips, 68881 and 68882. The compiler and related libraries optimize the code generation for each of these chips.

Better Performance Version 5.0 produces more compact code and faster code. We have seen as much as 10% performance improvements with the Version 5.0 system over the Version 4.0.

Global Optimizer Version 5.0 introduces our global optimizer. Some of the benchmarks, e.g. **Matrix**, execute up to 58% faster with the new global optimizer over the Lattice Version 4.0 release. The global optimizer's dead code elimination feature increases the performance of **Float** by over two orders of magnitude.

Full-Screen Debugger Version 5.0 introduces the CodePRobe (**CPR**) debugger. This debugger provides you with a full-screen symbolic interface to source code execution. CodePRobe supports advanced features like multitasking debugging.

Full-Screen Editor Version 5.0 includes the Lattice Screen Editor (**LSE**). This editor has been fully developed for the Amiga including gadget support and slider bars. It is fully programmable so that you can customize the keystrokes of LSE to that of your favorite editor. LSE also supports integrated development so that you can edit and compile from the same command set.

Programmer Utilities

Version 5.0 introduces numerous programmer utilities. These include:

Traceback	A utility that allows you to debug in the event of an abend.
Profiler	A utility that allows you to profile the execution of your application for performance tuning.
Build/Extract	Utilities that support command line generation.
CXREF	A C cross-reference facility.
DIFF	A utility that prints out the differences between two files.
fd2pragma	A pragma generator.
Files	A file manipulation utility.
Grep	A global search facility.
Compact	A header file compressor facility.
LMK	A powerful make file utility.
Splat	A global replace facility.
Touch	A file manipulation utility
WC	A word, sentence counting utility.

Compatibility

Version 5.0 is fully upward compatible with Version 4.0. Object modules and libraries created with either compiler may be freely mixed. As with any improvement, some of the newer features (registerized parameters and debugging support) will only be available to code compiled with the 5.0 compiler.

Library Improvements

Version 5.0 contains numerous library enhancements and new functions. Many of the library routines have been recoded to take advantage of better algorithms. Several performance sensitive routines have recoded into assembler to provide the tightest code. The new routines include **geta4**, **getenv**, and **putenv**.

ANSI Compliance

The compiler now supports the full ANSI preprocessor with string support, token support and appropriate

scoping of substitution symbols. The defined( ) directive is also supported. In addition, __DATE__ and __TIME__ provide the date and time of compilation. The compiler now uses sequence points to ensure correct evaluation and side effect generation according to the ANSI standard. Both the *const* and *volatile* keywords are supported. Function prototypes may now include an optional parameter name. Also functions taking a variable number of arguments may be indicated with ellipses '...'. String literals may now be concatenated to allow easier coding of long strings. *(void *)* correctly coerces a type without any warning. Many diagnostics have been added to allow detecting non-conforming programs.

New Keywords

The compiler now recognizes several new keywords:

Keyword	**Meaning**
signed	Overrides any default unsigned options.
near	Declares a data item to be addressed relative to the global base register. When used with a subroutine, it indicates a pc relative subroutine call.
far	Declares an item that must be addressed with a full 32 bit address.
huge	Same as far
chip	Declares a data item that must be placed in chip ram and addressed with a full 32 bit address.
__regargs	Defines a subroutine that is to be called with registerized parameters.
__stdards	Defines a subroutine that is to be called with standard stack parameters
__asm	Defines a subroutine that takes its parameters in a specific register
__SAVEDS	Defines a subroutine that is to load up the global base pointer upon entry.

__interrupt	Defines a subroutine that may be called from interrupt code.

Precompiled Headers

To provide for faster compilation of a large project, the symbol table obtained by compiling a program may be saved to disk. This precompiled header file may then be used as the starting point of another compilation to eliminate reparsing the header files encountered.

Ignore Multiple Includes

The compiler now supports an option to ignore multiple *#includes* of the same file. In addition, the search rules for *#include* files have been modified to conform to standard UNIX search conventions.

Error Messages

To allow more error messages to appear on the screen before scrolling off, we have adopted a more condensed format of displaying the error position in reverse video on the source line followed by the error message. This eliminates the visual confusion caused by the seemingly blank line between the message and the line. With the old format, there was a visual tendency to associate the error message with the wrong line.

Error Message Disabling

It is now possible to disable particular error messages as well as to change the severity of most messages. Along with this, it is now possible to specify a maximum number of errors allowed in a compilation so that the compiler will abort.

Improved Error Recovery

We have implemented an improved form of error recovery for many of the common mistakes to eliminate many of the situations that resulted in a cascade of errors.

Library Code Generation

The library base for #pragma statements is no longer limited to being a single external pointer. Any arbitrary expression may be used including function calls and local variables. This is intended to support generating code that resides in a library.

Big Compiler (LC1B)

In order to produce a compiler that fits well on a smaller system, we have elected to provide a "big" version of the compiler that includes some additional features. If you wish to take advantage of these features (at a cost of about 20K), you must use this big compiler instead of the standard one. The big compiler provides a full listing ability including macro expansion display, nest level counting, and include file listing. This listing may also include an optional cross reference of all variables, #define values and structure tags. The big version of the compiler may be used to generate prototype files of all functions encountered in a module. This eliminates the potentially tedious task of constructing the list of prototypes for all functions in a project.

Amiga library support

The compiler can generate code fully compatible with the requirement of an Amiga library routine - A6 is removed from the compiler selection list and A4 is made available for code use. Note that this code is incompatible with the **-b1** option.

Better control of code optimization

When generating code, you may now instruct the compiler to choose code sequences optimized for space or time depending on the type of application you may be building.

Registerized Parameters

Two styles of registerized parameters are supported. The **-rr** option causes the compiler to automatically place up to four parameters in registers for subroutine calls. The *__asm* keyword may be used in conjunction with a register specification list to cause the compiler to pass parameters in a given register.

International Support

Error messages are now in separate files for easy customizing to international error messages.

Resident Compatible

The compiler and tools are now multitasking and reentrant to support better use of the Amiga, and for faster load times. The compiler also builds resident compatible modules.

Acknowledgement

Lattice is grateful for the outstanding technical contributions to this product from SAS Institute.

New Features for 5.10

The SAS/C Lattice Amiga C Compiler Version 5.10 is an upgrade to the Lattice Amiga C Compiler Version 5.x. The major features for Version 5.10 include:

LSE AREXX Support	LSE, the editor, now has an AREXX interface. AREXX is a macro language developed for the Amiga by William Hawes. It is shipped with the 2.0 operating system, or available for 1.3 or 1.2 users by writing to
	`William Hawes`
	`P.O. Box 308`
	`Maynard, MA 01754`

LSE Creates Icons	LSE now has the ability to create icons for all files. See the section Starting a New Project.
LSE Screen Support	LSE supports 8- and 16-color Workbench screens in the 5.10 release.
Faster Linker	The linker, BLINK, is much faster in 5.10 - up to twice as fast, depending on your application. The speed increase is automatic and does not require you to change your scripts or LMK files.
Ability to Create Resident Libraries with BLINK.	It is now possible to generate resident libraies with BLINK quickly and easily, with little modification to your code.
LMK under Workbench	LMK can now be run from the Workbench by clicking on its icon or the icon of a lmkfile listing LMK as its tool. LMK and LC now launch processes at the same priority as the launching CLI, allowing you better control over your system's multitasking. Under the 2.0 operating system, LMK now allows longer command lines; under 1.3, it is still limited to 256 byte command lines due to system restrictions.

Default Options Default compile/link options are now read from an environment variable, or from a default file in your current directory. This allows you to specify your desired compile options in advance, then simply invoke the compiler to take advantage of them. A 2.0-compatible compiler preferences editor allows you set default options using a point-and-click interface. The editor runs under both 1.3 and 2.0. See the section **Utilizing SAS/C from the Workbench** for more information.

Improved Profiler The profiler, LPROF, now supports multiple code hunks. This means you are no longer required to use the SMALLCODE option to BLINK in order to use it.

EQUR support The assembler now accepts the EQUR directive.

__aligned LC1 now supports a new keyword, __aligned, which forces the object being declared to the next longword boundary. Since AmigaDOS requires many system data structures (like struct FileInfoBlock) to be longword aligned, this keyword can be used to force system data structures to a longword boundary so they can be declared static, extern or automatic instead of allocated dynamically. It must appear after the type specification, immediately before the variable name to be declared:

```
struct FileInfoBlock __aligned fib;
struct MYSTRUCT * __aligned myptr;
```

Multiple variables can be declared using the same declaration; all such variables will be aligned:

```
    struct FileInfoBlock __aligned fib1,
fib2, fib3;
```

You can align members of structures, but this may add hidden pad bytes to the structure:

```
struct MyStruct
{
   char a;
   struct FileInfoBlock __aligned fib;
/* Three bytes padding! */
};
```

You CANNOT align formal parameters:

```
/**** ILLEGAL! ****/
void func(char __aligned parm1, char
__aligned parm2);
/**** ILLEGAL! ****/
```

Prototype Generation

LC and LC1B's -pr option, which generates prototypes, now supports typedef names. If the routine for which you are generating a prototype uses the typedef name, the typedef name will appear in the generated prototype.

If you specify -pri, the identifier names will not be listed in the prototypes:

```
void func(int);
```

instead of

```
void func(int parm);
```

You can now specify the filename to which protypes will be written by using the -o option with -pr.

Preprocessor Limits

The size of the preprocessor expansion buffer can now be set on the command line. In the past, you would get a message like "preprocessor buffer overflow" or "preprocessor expansion too long or circular" if a macro expanded to more than a certain length. The new -z option on LC and LC1 allows you to specify any length

you like. Specify the size in bytes immediately after the -z, with no spaces:

```
lc -z4000 myfile.c
```

The default buffer size for lc1 is 3000 bytes; for lc1b, it is 6000 bytes. The minimum size is 512 bytes. Any specification less than 512 bytes will allocate 512 bytes.

Alternate Startups

When using LC -L to link your program after compiling, you can specify what startup code (c.o, cres.o, cback.o, catch.o, and so forth) by using the -t option on LC. The various forms of the -t option are:

Option	Startup
default	lib:c.o
-t	lib:cback.o
-tb	lib:cback.o
-tr	lib:cres.o
-tc	lib:catch.o
-tcr	lib:catchres.o
-t=<path>	<path>

The various startup objects do the following:

c.o (Default if no -t option present) Normal startup code. Initializes bss section, base pointer, library services, etc.

cback.o Detaches the process from the CLI and runs it in the background. See the description starting page G58-4.

cres.o Makes it possible to RESIDENT the program. This startup code makes a copy of the program's data section each time it is invoked, so each invocation

will get a different data section. Thus, all programs linked with cres.o will be 'pure' and can be placed on the resident list with the RESIDENT command. You must have less than 64k of external data to use cres.o.

catch.o, catchnr.o Produces a snapshot on software exceptions. Some software exceptions can be caught and handled by the program. With catch.o and catchnr.o, these execptions will be caught. catch.o will put up a requester when an exception occurs and ask you whether you want to write a snapshot file. catchnr.o will simply write the file without asking. The TB command, documented in the Commands section of the manual, will read the snapshot file and tell you what was going on when the program crashed.

catchres.o, catchresnr.o Combination of catch.o and cres.o. Clones the data segment in order to make the program pure, but it also catches exceptions. catchres.o puts a requester up on exceptions, catchresnr does not.

Auto near to far LC1 will automatically move near data to far if you attempt to declare too much near data. In the past, this would have resulted in a link error. This can happen if you are compiling without the -b0 switch, and you declare a large amount of static or external data.

no sizeof warning LC1 no longer issues an uninitialized variable warning if you use the construct

```
sizeof(*variable)
```

before you initialize 'variable'.

offsetof macros

Support for several varieties of the *offsetof* macro has been added or improved. *offsetof* takes two parameters, a structure and a member name, and returns the offset into the structure of the member. There are many ways of defining the offsetof macro. The compiler currently works with all the methods we know of, including those used by GNU and XWindows code. The simplest definition is as follows:

```
#define offsetof(type, name)
((long)&(((type *)0)->name))
```

C++ Comments

LC1 now supports C++-style line comments. They can be mixed with normal C /* comments. With // comments, you start the comment with the // characters, and the comment extends to the end of the line. All characters after the // are ignored.

```
i = 0;   // Initialize i
j = 10;  // Initialize j
```

New Warnings

Warnings have been added to LC1 to detect the following situations:

- If a nested comment is detected and comment nesting is disabled. (Warning 167)
- If an unbalanced comment is detected. (The line number of the beginning of the unbalanced comment is given.) (Warning 166)
- If an unbalanced #if/#ifdef is detected. (The line number of the beginning of the #if/#ifdef is given.) (Warning 139)

Comments and #if/#ifdefs are now considered to be unbounded if they have not been closed by the end of

the file in which they began. This means the message you get will list the file and line number with the error, not the end of the C source file.

Structure Equivalences

The -cq option was added to LC and LC1 in 5.04 and 5.05 to allow structure equivalence. If -cq is on, a structure A will be considered equivalent to a structure B if the types of struct A's fields match the types struct B's fields for the length of struct B. This means, for example, that you can pass a struct IntuiMessage to functions requiring a struct Message without casting. It also means you can pass an IOExtReq structure to functions requiring a struct IOReq, like SendIO(), DoIO() and so forth. Note that while a struct IntuiMessage is equivalent to a struct Message, a struct Message is *not* equivalent to a struct IntuiMessage, since it is shorter than the struct IntuiMessage.

Performance Improvements

Many changes have been made to LC2 to improve the generated code. The speed of LC1 has been improved, especially when generating debugging information with the -d option.

#pragma limit

The limit on the number of parameters to #pragma libcall has been raised to 14. This will allow ALL Amiga library functions to be called with parameters in registers.

GO ^C Checking

The global optimizer (GO) now checks for a user abort (control-c) at various points during its cpu-intensive processing. Previously, it only checked when doing I/O.

Minimum Stack Size

5.10 allows you to specify a minimum stack size by initializing the stack variable at compile time. If you do, and your program is invoked with a smaller stack, the startup code will allocate a new stack of the minimum size.

New Library Routines

The following UNIX® routines have been added to the libraries:

> opendir, readdir, seekdir, telldir, rewinddir, closedir (directory routines)
>
> stat (file information routine)
>
> isatty (test for terminal)

AmigaDos 2.0 Support

The product now includes the 2.0 include files, as well as the 1.3 includes.

User's Guide

User's Guide

Lattice Amiga C Compiler

Version 5.00

The C Development System for the Amiga

Lattice, Inc.
2500 S. Highland Avenue
Lombard, IL 60148
USA

A Subsidiary of SAS Institute Inc.

Lattice C Compmiler User's Manual

This manual was formatted using **HighStyle®** by Lattice, Inc.

Lattice® Amiga C Compiler

User's Guide

Part of the Lattice Amiga C Compiler Version 5.0

Lattice, Incorporated
2500 S. Highland Avenue
Lombard, IL 60148
USA

Subsidiary of SAS Institute Inc.

Lattice Amiga C Compiler User's Guide

This document was produced using ***HighStyle***™, the Lattice Document Composition System.

SAS/C® Compiler for AmigaDOS™

User's Guide

Version 5.10

Part of the SAS/C® Compiler for AmigaDOS™ Version 5.10

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C®C Compiler for AmigaDOS™User's Guide

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Table of Contents

APPENDICES

Section 1

Introduction

This is a user's guide for the AmigaDOS edition of the Lattice C Compiler. It explains how to install and operate the compiler, and it also discusses the C language as implemented by Lattice.

If you are already familiar with C and AmigaDOS, then you should have no problems using this guide. However, if you are new to C and/or AmigaDOS, you may find it useful to consult some of the publications listed in *Appendix A, Bibliography*.

1.1 Hardware Requirements

The Lattice compiler can be used on any Amiga computer having at least 512 kilobytes of main memory. Additional memory will be used if it is available. The system must also have either two diskette drives or one diskette drive and one hard drive.

We've found that the following configuration provides the greatest flexibility for our own programming staff:

- 512 kilobytes of main memory

- 2 megabytes of expansion memory
- Two 3.5-inch diskette drives
- One hard disk drive with a minimum capacity of 20 megabytes
- A color display
- A printer

Novice Amiga programmers often ask which to purchase first – the hard disk or additional memory. In our experience, the expansion of memory should always take precedence over a hard disk. While we are not suggesting that you expand your system to its maximum memory capacity of 8.5MB on the A1000 model or 9.5MB on the A500 or A2000 models, an additional 2.0MB seems to satisfy just about any programmer's needs.

Programs produced by the Lattice C Compiler have their own memory and I/O requirements. Your programs may need I/O devices that are not required by the compiler.

1.2 AmigaDOS Compatibility

The Lattice C Compiler and related utility programs are compatible with AmigaDOS Version 1.2, as are the programs produced by the compiler. Now that AmigaDOS is mature, new versions of the operating system seldom cause compatibility problems. However, if you install a new version of AmigaDOS and encounter difficulties with any Lattice product, contact our Technical Support Group. We will usually detect this situation before you do and will send out a customer bulletin describing the necessary workaround or update procedure.

1.3 Matters of Style

We have used a few fonts, icons, and other capabilities of our **HighStyle** publishing package in a way that we hope makes the manual easier to read. Here's a list of our major stylistic conventions:

1. Examples of AmigaDOS commands and C programs are set in a monospace font similar to that produced by the typical dot-matrix printer.

2. File names used within the narrative are set in italics, and case is not significant. That is, the file names *MYPROG* and *myprog* are equivalent, and so you can type them either way.

3. Italic text is also used when the narrative refers to a "dummy name" for which you are supposed to substitute the appropriate information. For example, in

   ```
   lc prog[↵]
   ```

   prog is a dummy name for which you should substitute the name of the C source file that you want to compile. The narrative text should make it clear whether an italicized word refers to a file name or a substitutable parameter.

4. Boldface type is used when the narrative refers to part of a command that must be typed as shown. For example, when discussing the command:

   ```
   lc -L myprog
   ```

   we'll show the option field as **-L** in the text to indicate that it should be typed exactly that way, while *myprog* is in italics to indicate that you are supposed to substitute the real program name for this dummy parameter.

5. We use the following symbols to represent some special keys on the Amiga keyboard:

   [↵] This is the RETURN key, which is also called the ENTER key. It is normally used to terminate each input line.

   [Ctrl][C] This key is called "control c" and is activated by holding down [Ctrl] and then pressing and releasing [C], with or without the shift key [Shift] depressed. This combination sends an

abort signal through AmigaDOS to the program that is currently running.

Ctrl \ This key is called "control backslash" and is activated by holding down Ctrl and then pressing and releasing \, the backslash key. If you are typing input data to a program, this combination indicates that you are finished. In other words, it is the keyboard end of file signal.

6. When we present a C programming example, we do not show the ENTER key ↵ at the end of each line, just to avoid the clutter. Even though C is a "free form language" and allows multiple statements on an input line, it's a good idea to keep the lines simple. So, when you build your source file with an editor, you'll normally press ↵ at the end of each line.

One final note on the style and contents of these manuals: *Please let us know if you see errors or omissions, or if you want to propose a better way of presenting certain information.*

Section 2

Installation

The Lattice Amiga C Compiler is distributed on diskettes using the Amiga 3.5-inch, double-sided format. These diskettes are not copy-protected and are set up for efficient use on a system that does not have a hard disk. However, we recommend that you NOT use the distribution diskettes directly. Instead, make a working copy, and keep the original distribution diskettes in a safe place.

Since the best performance will be obtained with a hard disk, the compiler package includes an installation program that makes it easy to transfer the software from the distribution diskettes to a hard disk.

2.1 Product Registration

If you purchased the compiler as a new product instead of an update, you must register with us in order to qualify for technical support and future updates. The *Customer Service Guide* included in the package explains the registration procedure. Please take a few minutes to register the product now, since it may save you some time in the future.

If you encounter any problems installing or using the compiler, contact the

Lattice Technical Support Group immediately. You can do this via telephone, TWX, FAX, BIX, or the Lattice bulletin board. These options are explained in the *Customer Service Guide*.

2.2 Standard Diskette System

If you do not have a hard disk, then the only thing you need to do before using the Lattice C Compiler is to make working copies of the original. This is easy if you are operating under the *Workbench* environment. Simply drag the icon of each distribution diskette onto the new disk's icon.

If the *Workbench* is not running – as indicated by the absence of disk icons on the right side of the screen after shrinking the CLI window – activate it by typing the following command:

```
loadwb↵
```

This command will place you in the *Workbench* environment where you can see the disk icons.

Another way to make working diskettes is via the DISKCOPY command, which is available when you are running under the command processor (CLI) instead of the *Workbench*.

The distribution diskettes and the working copies are bootable. In other words, diskette #1 can be used in place of your normal *Workbench* disk. After booting this diskette, the CLI environment is active, and the necessary environment variables are assigned, as described later. The *Workbench* icon-based environment is not loaded by this boot procedure. This saves about 100 kilobytes of main memory, thereby providing a larger space for the compiler and other programming tools. You can activate the Workbench via the *loadwb* command previously described.

2.3 Customized Diskette System

By just copying the distribution diskettes, you construct what we call a "standard diskette development system". It is also possible to set up a customized

diskette development system by re-arranging the Lattice files and combining them on diskettes with other programming utilities. *Appendix B, Diskette Contents* lists the files that are on the Lattice distribution diskettes.

This user's guide can't provide much general assistance with a customized system, but if you need some specific advice, contact the Lattice Technical Support Group.

2.4 Hard Disk System

To install the Lattice C Compiler on your hard disk drive, execute the *install_hd* procedure located in the S directory of diskette #1. See *Appendix B* for details on the contents of the distribution diskettes. This can only be accomplished from the CLI environment, which will be active if you boot diskette #1.

This procedure will create several subdirectories to contain the executables, header files, libraries, and source files. For example, if your working floppy disk is titled *'Lattice_C_5.0.1'* (where "1" is the disk number), then the command:

```
execute Lattice_C_5.0.1:s/install_hd↵
```

will perform the following actions:

- Create a directory named *sys:lc* and *sys:lc/c*
- Copy the executable files to *lc/c*
- Copy the header files to *sys:lc/include*
- Copy the libraries and object files to *sys:lc/lib*
- Copy the source files to *sys:lc/source*

The source file directory is not needed for normal compiler usage. It contains some example programs, the start-up routine, and several library routines that you may want to change. You may delete this directory if your hard disk is cramped for space.

2.5 Assigning Logical Devices

The Lattice C Compiler uses the following logical device names:

INCLUDE This name specifies the directory in which the compiler can find system header files, which are indicated by the use of angle brackets on the *#include* statement. For example, if you type the following assignment statement:

```
assign INCLUDE: dh0:lc/include↵
```

and then place the following *#include* statement in your C source file:

```
#include <stdio.h>
```

the compiler will attempt to include the file *dh0:lc/include/stdio.h*.

LC This name specifies where the Lattice compiler and other utility programs are located. For example,

```
assign LC: sys:lc/c
```

indicates that the compiler and utilities are in directory *sys:lc/c*. The default for this name is *sys:lc/c*.

LIB This name specifies where the linker can find the libraries and other modules needed to build executable files. If no assignment is made, the linker defaults to the directory *sys:lc/lib*.

QUAD This name specifies where the compiler should write its intermediate file, also known as a *quad file*. If no assignment is made, the compiler defaults to the current drive and directory. On a system without a hard disk, it is best to place the quad files on a different diskette drive that the source files. On any system, performance will be greatly improved if the quad files are placed in a RAM disk.

The appropriate assignment statements should be added to the file *s/startup-sequence* on your boot disk. For a typical installation on a hard disk, the statements are:

```
assign LC: sys:lc/c
assign INCLUDE: sys:lc/include
assign LIB: sys:lc/lib
assign QUAD: ram:
```

If your system that does not have a hard disk, you will usually boot from Lattice diskette #1, and so the assignment statements would typically be:

```
assign LC: Lattice_C_5.0.1:c
assign INCLUDE: Lattice_C_5.0.2:include
assign LIB: Lattice_C_5.0.2:lib
assign QUAD: ram:
```

Lattice diskette #1 is shipped with these statements in the setup file. However, you will need to change them if you distribute the various parts of the compiler package among your working diskettes in a different way.

Section 3

Compiler Operation

The Lattice Amiga C Compiler uses the classical *edit-compile-link* sequence for program development. In other words, you must first create a source file containing your C program. This is fed to the compiler, which produces an object file. Object files then serve as input to the AmigaDOS linker, which produces an executable file.

3.1 Creating Source Files

Source files are simply normal ASCII text files, consisting of text lines separated by newline characters. Most C programmers prefer a "full-screen editor" for source preparation, and many excellent products of this type are available. If you already own an editor with which you are comfortable, just continue to use it with Lattice C.

For those who do not own a full-screen editor, our compiler package includes the Lattice Screen Editor (LSE). In addition to its comprehensive editing capability, LSE can invoke the C compiler internally and automatically position the editing cursor on source lines that contain syntax errors, thereby making the edit-compile loop much more efficient. Complete operating instructions are provided in the *LSE User's Guide*.

For extremely simple programs, you can just copy the source lines from your keyboard to the source file, as shown in the following example:

```
cd ram:
copy * hello.c[↵]
#include <stdio.h>[↵]
void main() { printf("Hello world!\n"); }[↵]
[CTRL][\]
```

As mentioned in Section 1, the [CTRL][\] is called a "console end of file" and is generated by holding down [CTRL] while pressing the [\] key. This is the normal way to terminate input from the console.

3.2 Compiling and Linking One Module

Once you have created the source file *hello.c* as described above, you compile, link, and execute it via the following commands:

```
lc -L hello[↵]
hello[↵]
```

Before typing these lines, don't forget to set your logical device names correctly, as described in Section 2.

If everything works OK, then you will see "Hello world!" on your screen immediately after typing the last line. According to the venerable Kernighan and Ritchie book, you have now overcome the first hurdle on the path to learning a new language or compiler system.

Now let's examine what you did here. The first command

```
lc -L hello[↵]
```

invokes the compiler to process the file *hello.c* and produce an object file named *hello.o*. The **-L** field is an "option" specifying that *hello.o* should be linked with the Lattice C run-time system to produce the executable file *hello*. In other words, if the compilation is successful, the linker will be in-

voked to combine *hello.o* with the Lattice startup routine named *lib:c.o* and the two libraries named *lib:lc.lib* and *lib:amiga.lib*.

After this command, you should see some Lattice copyright messages on your screen, but you should not see any error or warning messages. If you do, then there is something wrong with the source program in *hello.c* or else you have not set the logical device names correctly.

The second command

```
hello⏎
```

executes the program named *hello* that was created by the linker. And of course, this program simply prints "Hello world!" to the screen and then terminates.

If you are not able to complete these steps with satisfactory results, you should contact Lattice Technical Support or consult with a friend who has more experience. We've found that first-time users often get hung up at this point and waste a lot of time because of simple "cockpit problems".

3.3 Using the Standard Math Library

When you are comfortable with the *Hello World!* program, try to compile and execute *ftoc*, which is one of the example programs installed in the source directory. Assuming that logical device LC: has been set correctly, this can be done by the following commands:

```
cd lc:/examples
lc -Lm ftoc
ftoc
```

This looks like the previous example, except that the **-Lm** option is used instead of **-L**. The **-Lm** form tells the linker to include the math library, which is necessary because *ftoc* performs floating point computations. The math library is actually named *lcm.lib*, and so when the linker sees **-Lm**, it searches *lib:lcm.lib* before searching the base library *lib:lc.lib*.

3.4 Using the Special Math Libraries

The Lattice C Compiler includes three math libraries:

LCM.LIB This is the standard math library which uses the IEEE format and contains all the mathematical functions described in the *Library Reference*.

LCMIEEE.LIB This library replaces some of the routines in the standard math library with versions that employ the IEEE routines supplied by Commodore on the Workbench diskette. If you want to use the Workbench routines, you must include this library before *lcm.lib* when linking a program that uses the IEEE format.

The Workbench routines are slightly slower than their counterparts in *lcm.lib* unless a math co-processor is present. In that case, the Workbench routines employ the co-processor to obtain a dramatic increase in floating point performance.

LCMFFP.LIB This library replaces most of the routines in the standard math library with versions that employ the Motorola Fast Floating Point (FFP) format. The FFP routines are faster than the IEEE routines, but they obtain this speed at the expense of accuracy. However, for many applications, the FFP accuracy is entirely adequate.

The compiler manipulates real numbers in the FFP format when you specify the **-f** option. Do not intermix the FFP and IEEE formats. In other words, if one module in an executable uses the **-f** option, then all modules must use it.

Also note that *lcmfpp.lib* does not contain all of the floating point functions that are in the standard math library. See the *Amiga ROM Kernel Manual* for a complete description of this library.

Let's look at *ftoc* again and show how you construct it with the special math libraries. First, here's how you would introduce the Workbench routines:

```
cd lc:/examples
lc -Lm+lib:lcmieee.lib ftoc
```

The option **-Lm + lib:lcmieee.lib** tells the linker to resolve external references by searching *lib:lcmieee.lib* first, followed by *lib:lcm.lib*, *lib:lc.lib* and *lib:amiga.lib*.

The following commands compile and link *ftoc* with the FFP number format:

```
cd lc:/examples
lc -f -L+lib:lcmffp.lib ftoc
```

The **-f** option tells the compiler to use the FFP format, and the **-L+lib:lcmffp.lib** option tells the linker to search *lib:lcmffp.lib* first, followed by *lib:lc.lib* and *lib:amiga.lib*. Notice that it is incorrect to use **-Lm + lib:lcmffp.lib**, because the standard math library uses the IEEE format instead of FFP.

3.5 Using Other Libraries

The **-L** option can be used to specify libraries other than those containing the math routines. The general rules are:

1. The **-L** option can be followed by one or more letters specifying libraries whose names begin with LC. For example, **-Lxpm** causes the linker to search the libraries named *lib:lcx.lib*, *lib:lcp.lib*, and *lib:lcm.lib*.
2. The **-L** option can be followed by a list of libraries separated by plus signs (+). For example, **-L+lib:mylib.lib+df0:yourlib.lib** causes the linker to search the libraries named *lib:mylib.lib* and *df0:yourlib.lib*.
3. If the two forms discussed above are used together, the single-letter libraries are searched last. For example, **-Lxpm+lib:mylib.lib+df0:yourlib.lib** causes the linker to search *lib:mylib.lib*, *df0:yourlib.lib*, *lib:lcx.lib*, *lib:lcp.lib*, and *lib:lcm.lib*.

4. The basic runtime support library, *lib:lc.lib*, and the AmigaDOS binding library, *lib:amiga.lib*, are always searched last.

With these rules understood, you might want to change the names of the special math libraries in order to simplify their use. For instance, you could rename *lcieee.lib* to *lcw.lib*, where the "w" indicates that this is the Workbench interface. Then you could change *lcffp.lib* to *lcf.lib*. To compile and link *ftoc* with the Workbench library, the commands are:

```
cd lc:/examples
lc -Lwm ftoc
```

Note that you must write **-Lwm** instead of **-Lmw** so that the Workbench library is searched first. To compile and link with the FFP library, use these commands:

```
cd lc:/examples
lc -f -Lf ftoc
```

3.6 Using the Linker

In many situations it is more appropriate to call the linker directly instead of using the **-L** option on the LC command. Here is how you would compile and link *ftoc* in this way:

```
cd lc:/examples
lc ftoc
blink lib:c.o,ftoc.o to ftoc lib lib:lcm.lib,lib:lc.lib,lib:amiga.lib
```

You must specify the startup routine, *lib:c.o*, as the first object module. Also, the libraries *lib:lc.lib* and *lib:amiga.lib* must be the last ones searched. More details about the BLINK linker can be found in the *Command Reference*.

3.7 Using the Global Optimizer

In order to use the global optimizer capability of the compiler, you use the **-O** option on the compile command. Here is how a compile would look with the global optimizer being used.

```
cd lc:/examples
lc -O ftoc
```

Additional time will be required to complete your compilation when the global optimizer is being used.

Library Summary for 5.10

Listed below are descriptions for each of the Lattice Libraries included with the 5.10 package. We have named them based on the following scheme.

- If there is an 'm' in the name, it is a math library and contains floating point routines.
- If there is an 's' in the name, then the library is to be used with the 'short integer' compiler option -w.
- If there is an 'r' in the name, then the library is to be used with registerized parameters, -rr option.
- If the name contains the sequence 'nb', then the library does not reference a base register, all data is reached by absolute addressing. Use these libraries with -b0.
- The math libraries may have one of three extensions.

Extension	Usage
None	Lattice IEEE format.
IEEE	Uses commodore IEEE routines.
FFP	Motorola Fast Floating Point format.
881	Uses in-line 68881 instructions.

> To get the best performance with a 68881 library, include the m68881.h header file, and compile with the -f8 option.

Listed below are the specific libraries and their descriptions. Note that not all possible combinations are supported.

Library	Usage
lc.lib	Lattice Standard C Library.
lcr.lib	Lattice C Library for use with Registerized Parameters.
lcs.lib	Lattice C Library for use with 16-bit integers.
lcnb.lib	Lattice C Library for use with no base-relative data addressing.
lcsr.lib	Lattice C Library for use with 16-bit integers and Registerized Parameters.
lcsnb.lib	Lattice C Library for 16-bit integers and no base-relative addressing.
lcm.lib	Lattice Standard IEEE Math Library.
lcmr.lib	Lattice IEEE Math Library for use with Registerized Parameters.
lcms.lib	Lattice Standard IEEE Math Library for use with 16-bit integers.
lcmffp.lib	Lattice Motorola Fast Floating Point Math Library.
lcm881.lib	Lattice 68881 Coprocessor Math Library.
lcmieee.lib	Lattice IEEE Math Library for use with Commodore Resident Library.
amiga.lib	Library of linkage routines to Amiga Resident Libraries.
ddebug.lib	Commodore debug library for use with Parallel Port.
debug.lib	Commodore debug library for use with Serial Port.

Utilizing SAS/C from the Workbench

With the 5.10 update, we have added the ability to readily access the SAS/C compiler environment from the Workbench without having to resort to using the CLI. Even the more sophisticated programmers will wish to take advantage of this environment, as it eliminates much unnecessary typing, replacing it instead with a mouse click.

The most important aspect of this environment to understand is that a project consists of all the source files that are found in a single directory. For future releases, this limitation will be eliminated. Additionally, work is accomplished through a few additional project icons that are placed in the work directory.

To create a project directory (or update an existing one to support this environment), put copies of the STARTER PROJECT icons into your directory. Under the CLI, this can be done with the command:

```
copy lc:/STARTER PROJECT/#? mydir clone
```

With the Workbench it is slightly more complicated because the Workbench copies icons differently when moving between disks versus on the same disk.

If your new project directory is to be on the same disk as the STARTER PROJECT directory, you need to:

- Select the STARTER PROJECT drawer
- Choose the DUPLICATE item from the Workbench menu
- Select the "Copy of STARTER PROJECT" drawer created
- Choose the RENAME item from the Workbench menu
- Press Right-Amiga-X to erase the contents of the requester and then type in the name of what you want to call the new project. Press the Return key when you have entered the new name.
- Drag the renamed DRAWER to the desired target location.

If the new project directory is to be on a different disk, you can do this more simply by:

- Dragging the STARTER PROJECT drawer to the desired target disk. If you do not have two drawers with the same name after this operation then you know that the target location is on the same disk. If this happens, simply drag the drawer back.
- Choosing the RENAME item from the Workbench menu
- Pressing Right-Amiga-X to erase the contents of the requester and then type in the name of what you want to call the new project. Press the Return key when you have entered the new name.

Once you have completed this operation, you will have a project drawer with at least three icons:

ICON	Function
OPTIONS	Runs the SAS/C Options to set compiler options
EDIT	Runs LSE to edit a new file
BUILD	Runs LMK/LC to compile and link your project

> Note that Workbench does not always notice when new icons are created - particularly from the CLI. If an icon fails to show up in the drawer window, simply close the window and reopen it. You will have to do this whenever you create a new file with the editor.

The first stage is to run the SAS/C Options program by clicking on the OPTIONS icon. It can also be run from the CLI by typing sascopts. The program will open a window with gadgets for controlling the basic compiler options. Astute observers will immediately notice that this program implements the AmigaDos 2.0 look and feel - even under 1.3.

In the window will be five basic types of gadgets:

Cycle The gadgets appear as a raised rectangular button with a cycle symbol on the left hand side. By clicking on the button with the

left mouse button, you can cycle to the options available. Pressing the shift key at the same time will reverse the direction of the cycle.

Check These gadgets appear as text to the right of a small raised rectangular button. Selecting the button will cause a check mark to appear and disappear. When the check mark is visble, the option is selected.

Action These gadgets appear as raised buttons with text on them. Selecting the button will cause an immediate action to occur.

Strings These gadgets appear as a raised ridge around a text area. You can enter data in the area by clicking within the text area and typing. When you are satisfied with the entered data, press return.

Lists These appear as a complex set of control gadgets combined with a raised scrolling area. The slider and arrow gadgets are used to position the scrolling area on a particular data item. Clicking on an item in the scrolling area will cause it to appear on the String area for modification. This selected item may be removed from the list by selecting the DEL button. New items may be added to the list by selecting the NEW button.

Together the gadgets provide a visual interface to setting almost all of the LC compiler options. The initial window contains the most commonly used options. Adventuresome programmers may also use the advanced options and object options to provide finer control over compiling.

The options provided on the primary screen are:

Gadget Text	Type	Equivalent LC options
Debug Level	Cycle	-d -d1 -d2 -d3 -d4 -d5
Data Model	Cycle	-b1 -b0
Code Model	Cycle	-r0 -r1
Floating Point	Cycle	-fi -fl -ff -f8
Machine Code	Cycle	-m0 -m1 -m2 -m3 -m4 -ma
Calling Mode	Cycle	-rs -rr -rb
Strict Ansi	Check	-ca
Global Optimizer	Check	-O
Short Integers	Check	-w
No Stack Checking	Check	-v
Tiny Link Options	Check	-Lt
Require Prototypes	Check	-cf
Single String Copy	Check	-cs
Unsigned Characters	Check	-cu
Quiet int return	Check	-cw
Include Directories	List	-i
#define symbols	List	-dx=y
Program Name	String	-p<name>

By selecting the **Advanced Options** gadget on the primary screen, you can get access to some additional compiler options. On the Advanced screen, you will find:

Gadget Text	Type	Equivalent LC options
Nested Comments	Check	-cc
$ in Identifiers	Check	-cd
no Error Line	Check	-ce
No Multi-Includes	Check	-ci
Allow chip/near/far	Check	-ck
Multi-Char Constant	Check	-cm
Disallow __asm	Check	-cr
Warn Undefined Tags	Check	-ct
8 Char Identifiers	Check	-n
Structure Passing	Check	-c+
List Source	Check	-gs
List Macros	Check	-gm
List Excluded Lines	Check	-ge
List Included Files	Check	-gi
List Header Files	Check	-gh
XREF	Check	-gx
XREF Define Symbols	Check	-gd
XREF Compiler Files	Check	-gc
Undefine Symbols	Check	-u
Error Limit	String	-q<n>e
Warn Limit	String	-q<n>w
Precompiled Headers	List	-h
Error/Warning Control	List	-j

From the primary screen, you can get access to object generation options by selecting the **Object Options** gadget. On this screen, you will find:

Gadget Text	Type	Equivalent LC options
Startup Selection	Cycle	-tr -t -tc -tcr -t=
Code Location	Cycle	-ac -hc
Data Location	Cycle	-ad -hd
BSS Location	Cycle	-ab -hb
Optimization Target	Cycle	-mt -ms
No Auto-Register	Cycle	-mr
Default Segments	Check	-s
Load A4 at entry	Check	-y
Disable Call Merge	Check	-mc
ADDSYM	Check	-La
SMALLCODE	Check	-Lc
SMALLDATA	Check	-Ld
VERBOSE	Check	-Lv
NODEBUG	Check	-Ln
Map Hunk	Check	-Lh
Map Symbols	Check	-Ls
Map Libraries	Check	-Ll
Map Xref	Check	-Lx
Map Overlay	Check	-Lo
Linker Objects	List	-L+
Make Library	String	-r<name>
User Startup	String	-t=<name>
Code Name	String	-sc=<name>
Data Name	String	-sd=<name>
BSS Name	String	-sb=<name>

A more complete explanation of these options can be found on pages C40 thru C59.

Once you have selected the options, you may choose to save them as the project options by selecting the SAVE button. If you wish to set up the options as a global default (but always overridden by any local option setting) you can select the SAVE DEFAULT button. It will store the options in the ENV:sascopts variable. You can exit without making any changes by selecting the CANCEL button.

Menu items are also available for reading in default options and saving to a specific location.

The only option that must be filled in is the Program Name string field. Without this, the compiler and linker will not know what to call the final program that they create. You can even save this name as the global default so that all programs you create will have the same name although this is not recommended.

Once the options are chosen, the next step is to edit and create the program to be compiled and run. The provided LSE editor will run from the Workbench and make icons for any files that it creates. You are not tied to using the LSE editor - any editor that can run with Workbench in this manner will suffice.

Once you have created the appropriate source files, just double click on the BUILD icon. This will cause LMK to run and build your program using the default options. Note that in doing so, LMK will compile all C source files in the current directory and link them together using the name supplied in the options.

If there are any errors, LMK will stop and put up a requester indicating so. Any errors should be visible in the LMK window. To correct them, just invoke the editor on the appropriate files and fix them.

Once you have successfully built the project, you may run it by double clicking on the icon that Blink creates for the program.

Programs invoked from the workbench receive a different environment than when invoked from a CLI. In order to properly work in a Workbench environment, your program must be sensitive to how it was started.

You can customize the program that is built through the use of an LMKFILE. If one is present in the current directory, LMK will utilize it for the building rules instead of compiling and linking all source files in the directory. Pages U55 through U86 explain how to construct and maintain an LMKFILE.

Note that this environment is not yet a complete integrated environment. We recognize that there are many steps to improving it but felt it important to provide a mechanism for Workbench users to get to the power of the SAS/C compiler. Future developments will greatly refine and improve upon the concept.

Section 4

Language Definition

This section describes the C language as supported by Version 5 of the Lattice Amiga C Compiler, which conforms closely to the compilers associated with UNIX System V. The ANSI committee for C standardization has chosen to use the UNIX V definition as their starting point. The Lattice compiler also conforms to almost all of the ANSI standard. The goal is to have complete compilance.

Of course, the committee has also proposed several additions to the language, and Lattice is actively participating in this refinement process as a committee member. The most important of these additions, a feature called "argument type checking", has been incorporated into the Lattice compiler, as has the new *const* data type. The others will be added as they become better defined.

Notice that we are not providing a complete C language specification with the Lattice C Compiler. *Appendix A* cites several books that do this very well, including the Kernighan and Ritchie (K&R) text which began the popular movement towards C. We recommend that you obtain K&R plus one of the more recent C programming books that discuss the latest UNIX and ANSI language features.

In this section, we'll assume that you have a general knowledge of the language obtained from such books and/or from actual experience with another C compiler. Then we'll describe the specific characteristics of the Lattice compiler, which fall into three categories:

1. How does the C language as supported by Lattice differ from the specification provided in K&R? We've chosen to use K&R as our basis for comparison simply because the document describing ANSI's proposed C standard is not yet in the hands of most C programmers.
2. How does Lattice C handle the murky areas of the language that are usually written off as "implementation issues"?
3. What are the processing limits of the Lattice compiler?

Also, if you have been using an earlier version of Lattice C, you will find Version 5 to be somewhat different, both in language features and in library features. These differences are highlighted in *Appendix D.*

4.1 Comparison to K&R

For many years the most precise definition of the C programming language generally available has been the Kernighan and Ritchie book entitled *The C Programming Language.* This book contains an appendix entitled *The C Reference Manual,* which we abbreviate as CRM.

CRM defines the C language as it is most widely known today, but since CRM does not include many of the changes introduced by ANSI, it is gradually becoming a subset definition. A second edition of K&R was recently published to reflect the ANSI-inspired changes to the language. Nonetheless, we continue to use the original edition as our basis here simply because it is the book with which most C programmers are familiar. Both editions are cited in *Appendix A* of this user's guide.

The following list highlights the differences between Lattice C and the original K&R definition. The items use the K&R numbering scheme. For example, **CRM 2.1 Comments** refers to paragraph 2.1 in the K&R C Reference Manual.

CRM 2.1 Comments

Although the default mode is that comments do not nest, the **-cc** option can be used to allow nested comments. Here's an example of nested comments:

```
/*

Outer Comment Block

/*

Inner Comment Block

*/

Outer Comment Block

*/
```

CRM 2.3 Keywords

Lattice C includes several additional keywords: *const*, *enum*, *void*, and *volatile* which are ANSI-compatible, and *chip*, *far* and *near*, which are non-ANSI extensions.

CRM 2.4.1 Integer constants

Names declared as values for an enumeration type may be used as integer constants.

CRM 2.4.3 Character constants

Two new escape sequences are recognized:

\v	Specifies a vertical tab (VT) character.
\x	Introduces one or two hexadecimal digits which define the value of a single character. For example, '\xf9' generates a character with the value 0xF9.

Although by default the compiler permits single-character constants (e.g. 'A'), the **-cm** option enables multi-character constants containing up to four characters (e.g. 'AB', 'ABC', 'ABCD'). Such a constant is a short integer for two characters and a long integer for three or four characters.

CRM 2.5 Strings

The same **\x** convention described above can be employed in strings as in "ABC\xF9DE".

In addition, the **-cs** option causes the compiler to recognize identically written string constants and only generate one copy of the string. Note that a quoted string used to initialize a character array is not actually treated as a string constant, since it is actually placed into the array at compile time. For example,

```
char abc[] = "1234";
char *p = "1234";
```

produces a 5-byte array named **abc** containing the digit characters from 1 through 4, followed by a null byte. The second declaration produces a string constant similar to the contents of **abc** and places a pointer to the constant into **p.**

CRM 4. What's in a name?

const The *const* type should be used to declare an initialized data item that will never change. For example,

```
char const name[] = "12345abc";
```

declares a constant string.

far The *far* type indicates that the object must be accessed with a 32-bit address.

enum The *enum* type should be used to declare an integral item that can only have certain named values, each of which is treated as an integral constant. The actual values assigned to the identifiers normally begin at zero and are incremented by one for each successive identifier. However, an explicit value can be forced by using an equal sign, and then subsequent identifiers are assigned the new value plus one, etc.

For example, this statement defines an *enum* type:

```
enum color
  {
  red,
  blue,
  green=4,
  puce,
  lavender
  };
```

and this one defines some objects of that type:

```
enum color x, *px;
```

In this example, the symbols associated with the enumerated type *color* are given the following values:

```
0 => red
1 => blue
4 => green
5 => puce
6 => lavender
```

Each enumeration is a separate type with its own set of named values. The properties of an *enum* type are identical to those of *int* type.

near The *near* type indicates that the object may be accessed with a 16-bit pointer.

void The *void* type should be used when declaring a function that has no return value. If a function is not declared as *void* and does not appear to return a value, the compiler will issue a warning message.

volatile The *volatile* type indicates that the object may be changed by forces outside of the current program. Typical examples of this are memory-mapped I/O registers and shared memory in a multi-tasking environment.

CRM 6. Conversions

An expression can be converted to the *void* type by means of a cast. This is often used to explicitly indicate the discarding of a function return value. An expression of type *void*, however, cannot itself be converted or used in any way.

CRM 7.1 Primary expressions

The Lattice compiler always enforces the rules for the use of structures and unions so that it can determine which set of member names is intended. Since the compiler maintains a separate set of member names for each structure or union, the primary expression preceding a period (.) or arrow (->) operator must be immediately recognizable as a structure or as a pointer to a structure of the type that contains the specified member name. For example, given these declarations:

```
struct A
  {
  int x;
  int y;
  } m,*p;

struct B
  {
  int xx;
  int yy;
  } n,*q;
```

the following statements would be invalid:

```
p = &n;     /* p must point to structure type A */
q = &m;     /* q must point to structure type B */
```

because the pointers do not refer to the correct structure types.

You can, of course, convince the compiler that you really want to generate code for these statements by using the appropriate cast operations, as follows:

```
p = (struct A *)(&n);
q = (struct B *)(&m);
```

CRM 7.2 Unary operators

The requirement that the ampersand (&) operator can only be applied to an lvalue is relaxed slightly to allow application to an array name (which is not considered an lvalue). Note that the meaning of such a construct is a pointer to the array itself, which is quite different from a pointer to the first element of the array.

The difference between a pointer to an array and to an array's first element is only important when the pointer is used in an expression with an integral offset, because the offset must be scaled (multiplied) by the size of the object to which the pointer points. When the pointer points to the array instead of to the first array element, the target object size is the size of the whole array, rather than the size of a single element.

CRM 7.7 Equality operators

The only integer to which a pointer may be compared is the integer constant zero. You can, of course, cast any integer into a pointer. For example, the *sbrk* function normally returns a pointer, but it returns a value of -1 in the event of an error. This is a UNIX gaffe, but you can still perform a syntactically correct error check in the following way:

```
char *p;

p = sbrk(1000);                     /* get 1000 bytes */
if(p == (char *)(-1)) abort(3);     /* error check */
```

CRM 7.14 Assignment operators

Both operands of the simple assignment operator (=) may be structures or unions of the same type.

CRM 8.1 Storage class specifiers

The K&R text states that the storage class specifier, if omitted from a declaration outside a function, is taken to be *extern*. This is an erroneous statement, which K&R clarifies in CRM 11.2. The presence or absence of *extern* on a declaration outside of the scope of any function is critical to determining whether an object is being defined or refer-

enced. If *extern* is present, then the declared object either exists in some other file or is defined later in the same file. But if no storage class specifier is present, then the declared object is being defined and will be visible in other files. If the *static* specifier is present, the object is defined but is not made externally visible.

The only exception to these rules occurs for functions, where it is the presence of a defining statement body that determines whether the function is being defined. For example, the following statements are equivalent:

```
extern int foo();
int foo();
```

Both statements reference function *foo* rather than define it. The following statement, however, defines the function:

```
foo()
  {
  printf("Hello\n");
  return(0);
  }
```

It would be erroneous to use the *extern* keyword in front of this declaration.

The Lattice compiler can be forced to assume *extern* for all data declarations outside a function by means of the **-x** compile time option. Declarations which explicitly specify *static* or *extern* are not affected.

CRM 8.2 Type specifiers

Three new type specifiers are supported, as mentioned earlier: *const*, *enum*, and *void*. Also, the compiler recognizes the following new types that are specified via multiple keywords:

```
unsigned char
unsigned short
unsigned short int
unsigned long
unsigned long int
```

CRM 8.5 Structure and union declarations
The Lattice compiler maintains a separate list of member names for each structure and union. Therefore, a member name may not appear twice in a particular structure or union, but the same name may be used in several different structures or unions within the same scope.

Also, structure and union tags are in the same class, which means that you cannot use the same tag for both a structure and a union.

Enumerations are declared in much the same way as structures and unions, with a separate list of identifiers for each enumeration type. Enumerations are unique types which can only assume values from a list of named constants. The language treats them as *int* values but restricts operations on them to assignment and comparison. The named constants, however, may appear wherever an *int* is legal.

The optional name which may follow the keyword *enum* plays the same role as the structure or union tag; it names a particular enumeration. All such names share the same space as structure and union tags. The names of enumerators in the same scope must be distinct from each other and from those of ordinary variables.

CRM 8.7 Type names
Although a structure or union may appear in a type name specifier (i.e. a cast), it must refer to an already known tag, that is, structure definitions cannot be made inside a type name. Thus, the sequence:

```
(struct { int high, low; } *) x
```

is not permitted, but

```
struct HL { int high, low; };

(struct HL *) x
```

is acceptable.

CRM 10.2 External data definitions
The Lattice compiler applies a simple rule to external data declara-

tions. If the keyword *extern* is present, the compiler assumes that the actual storage will be allocated elsewhere, and so it generates a reference to the storage area which will be resolved by the linker. Otherwise, the declaration is interpreted as an actual definition which allocates storage, unless the **-x** option has been used, as described previously under CRM 8.1.

CRM 12.3 Conditional compilation

The constant expression following *#if* may not contain the *sizeof* operator and must appear on a single input line. Also, the *#elif* (else if) is supported along with the *defined( )* operator so that you can use logical operators to string *#if* name conditions together on a single line. The following is an example of a multicondition *#if.*

```
#if defined(name1) | !defined(name2)
```

CRM 12.4 Line control

Although the filename for *#line* must be an identifier, it need not conform to the characteristics of C identifiers. The compiler takes whatever string of characters is supplied, and the only lexical requirement for the filename is that it cannot contain any white space.

CRM 14.1 Structures and unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions.

The escape from typing rules described in K&R is explicitly not allowed by the Lattice compiler. In a reference to a structure or union member, the name on the right must be a member of the aggregate named or pointed to by the expression on the left. Our implementation, however, does not enforce any restrictions on references to union members, such as requiring a value to be assigned to a particular member before allowing it to be examined via that member.

4.2 Compiler Implementation Decisions

This section describes how the Lattice compiler deals with some aspects of the C language that have fallen into the domain of "implementation decisions". Some of these issues were left up to the compiler designer simply through oversight in the language definition, and the ANSI committee is gradually closing these loopholes. Others cannot be resolved in the language definition because they depend upon the specific hardware or software limitations faced by the compiler designer.

4.2.1 Pre-Processor Features

1. The Lattice C compiler supports the full set of pre-processor commands described in K&R. Pre-processor commands are handled concurrently with lexical and syntactic analysis of the source file, because there is no requirement to have a separate pre-processor pass, and compiler performance is improved by this approach. Nonetheless, analysis of the pre-processor commands is largely independent of the compiler's C language analysis. For example, *#define* text substitutions are not performed for pre-processor commands, but nesting of macro definitions is possible because substituted text is re-scanned for new *#define* symbols.

2. Since the compiler uses a text buffer of fixed size, a particularly complex macro may occasionally cause a line buffer overflow condition. Usually, however, this error occurs when there is more than one macro reference in the same source line, and it can be circumvented by placing the macros on different lines.

3. Circular definitions such as:

   ```
   #define A B
   #define B A
   ```

   will be detected by the compiler if either A or B is ever used. More subtle loops are also detected.

4. Like many other implementations of C, the Lattice compiler pushes macro definitions onto a stack, so that if the line:

   ```
   #define XYZ 12
   ```

   is followed later by:

   ```
   #define XYZ 43
   ```

   the new definition takes effect, but the old one is not forgotten. In other words, after encountering:

   ```
   #undef XYZ
   ```

   the former definition (12) is restored. To completely undefine XYZ, an additional *#undef* is required. The rule is that each *#define* must be matched by a corresponding *#undef* before the symbol is truly forgotten. Also, the compiler issues a warning message whenever a macro is re-defined. This was done to help detect those nasty errors that occur when header files are in conflict.

5. Two clarifications should be noted with regard to the *#if* command. First, an undefined symbol in a *#if* expression is treated as having a value of zero. Second, a symbol defined with null substitution text is interpreted as having a value of one. These conventions are consistent with *#ifdef* usage, and permit the use of expressions like:

   ```
   #define SYM1
   #define SYM2 0
   #if SYM1 | SYM2 | SYM3
   ```

   which causes subsequent code to be processed because SYM1 is treated as if its value were one.

4.2.2 Scope of Identifiers

The Lattice compiler conforms almost exactly to the scope rules discussed in

CRM 11. The only exception arises in connection with structure and union member names, where, in accordance with later versions of the language, the compiler keeps separate lists of member names for each structure or union. Two other points are worth clarifying.

First, the compiler does not generate specific allocate and de-allocate instructions for *auto* items declared in statement blocks within a function. Instead, the function entry sequence allocates enough stack storage to handle the largest collection of automatic data items so declared. With this scheme, a function may allocate more stack space than is actually used, but the need for run-time dynamic allocation within the function is avoided.

Second, when an identifier with a previous declaration is redefined locally as an *extern*, the previous definition is superseded, but the compiler also verifies compatibility with all preceding *extern* definitions of the same name. This is done in accordance with the standards, which require that all references to the same external name must be to the same object. The point is that in this particular case, where a local block redefines an identifier as extern, the local declaration does not actually disappear upon termination of the block because the compiler now has an additional external item for which it must verify later declarations.

4.2.3 Initializers

Objects which are of the static storage class are guaranteed to contain binary zeros when the program begins execution, unless an initializer expression is used to define a different initial value. The Lattice compiler supports the full range of initializer expressions described in Kernighan and Ritchie, but restricts the initialization of pointers somewhat:

1. A pointer initialization expression must evaluate to the integer constant zero or to a pointer expression of exactly the same type as the pointer being initialized. This pointer expression can include the address of a previously declared static or external object, plus or minus an integer constant. However, it cannot contain a cast operator applied to a variable, because such conversions cannot be done at compile time.

2. This restriction makes it impossible to initialize a pointer to an array unless the & operator is allowed to be used on an array name, because the array name without the preceding & is automatically converted to a pointer to the first element of the array. Accordingly, the Lattice compiler accepts the & operator on an array name so that declarations such as:

   ```
   int a[5], (*pa)[5] = &a;
   ```

   can be made. Note that if a pointer to a structure (or union) is being initialized, the structure name used to generate an address must also be preceded by the & operator.

3. An arithmetic object may be initialized with an expression that evaluates to an arithmetic constant which, if not of the appropriate type, is converted to that of the target object.

4. More complex objects (arrays and structures) may be initialized by bracketed, comma-separated lists of initializer expressions, with each expression corresponding to an arithmetic or pointer element of the aggregate. A closing brace can be used to terminate the list early. See Appendix A of Kernighan and Ritchie for examples.

5. Unions may also be initialized like structures. The initialization list must correspond to the first member of the union.

6. A character array may be initialized with a string constant which need not be enclosed in braces. This is the only exception to the rule requiring braces around the list of initializers for an aggregate.

7. Initializer expressions for auto objects can only be applied to simple arithmetic or pointer types (not to aggregates), and are entirely equivalent to assignment statements.

4.2.4 Expression Evaluation

All of the standard operators are supported by the Lattice compiler, using the standard order of precedence as described on K&R page 49. Expressions

are evaluated using an operator precedence parsing technique which reduces complex expressions to a sequence of unary and binary operations involving at most two operands.

Operations involving only constant operands (including floating point constants) are evaluated by the compiler immediately, but no special effort is made to re-order operands in order to group constants. Thus, expressions such as:

```
c - 'A' + 'a'
```

must be parenthesized so that the compiler can evaluate the constant part:

```
c + ('a' - 'A')
```

If at least one operand in a binary operation is not constant, the intermediate expression result is assigned to a temporary storage location. The temporary then replaces the binary operation in the expression and becomes an operand of another binary or unary operation. This process continues until the entire expression has been evaluated.

The use of temporaries is optimized by the compiler so as to minimize regeneration of identical temporaries during a straight-line code sequence. Thus, common sub-expressions are recognized and evaluated only once. For example, in the statement: /

```
a[i+1] = b[i+1];
```

the expression *i+1* will be evaluated once and used for both subscripting operations. This same optimization strategy eliminates expressions producing useless results with no other side effects, such as:

```
i+j;
```

Three conditions cause temporaries to "die" during a straight-line code sequence:

1. A function call may have side effects which cause previously-computed temporaries to no longer be accurate, and so all temporaries are discarded after the call.
2. If either operand associated with a temporary is the result of a later operation, the temporary is discarded after that operation.
3. When the result of an operation is stored through a pointer, the compiler discards all temporaries constructed from operands having the same type as the pointer. This is necessary because the compiler cannot determine if the pointer refers to a component of a current temporary value, and so it takes the safe approach.

   Note that this strategy may fail if the programmer uses "type punning". For example, if variable A is declared as an integer and you construct a character pointer that, in fact, points to A's storage location, the compiler will not discard temporaries containing A when you change A via the character pointer.

Except for common sub-expression detection, which may replace an operation with a temporary value, expressions are evaluated in left-to-right order unless that is prevented by operator precedence or parentheses. However, it is best not to make any assumptions about the order of evaluation, since the language definition allows C compilers to re-order expressions in order to generate better code. We may introduce this type of optimization in a future release.

Note that the language definition does guarantee that logical OR (||) and logical AND (&&) operators will be evaluated in left-to-right order.

4.2.5 Control Flow

C offers a rich set of statement flow constructs, and the Lattice compiler supports the full complement of them. Since control flow operations tend to dominate many programs, the compiler takes special steps to optimize them, as follows:

1. Switch statements are analyzed to verify that they contain
   - At least one case or default entry;

- No duplicate case values;
- Not more than one default entry;

Then the compiler chooses the best of several machine code sequences to test for the various cases.

2. The size and number of branch instructions is kept to a minimum by extensive analysis of the flow within each function.
3. Unreachable code is discarded, with the appropriate warning messages.

4.3 Compiler Limitations

To conclude this description of the Lattice's C compiler implementation, we list the more important internal limits of the compiler.

1. The maximum value of the constant expression defining the size of a single subscript of an array is two less than the largest unsigned target machine int (65,533 for a 16-bit int, 4,294,967,295 for a 32-bit int).
2. The maximum length of an input source line is 512 bytes.
3. The maximum size of a string constant is 256 bytes.
4. Macros may have no more than 16 arguments.
5. The maximum length of the substitution text for a *#define* macro is 512 bytes.
6. The maximum level of *#include* file nesting is 16.

These limitations have proven to be adequate for most C programs, and they also lead to very good compiler performance.

Section 5

Programming Environment

Programming with the Lattice C Compiler is not much different than programming in C on a UNIX system, which is where the C language originated. This is an important statement, because it implies that you can use just about any of the popular C programming books with the Lattice compiler. *Appendix A* lists some of the publications that you may want to consider.

Now, if you confine your programming to the features that most of these books describe, you really don't need to know much at all about the AmigaDOS environment. However, if you will be doing special things such as linking with assembly-language functions or writing I/O drivers in C, then you will need to know more about the low-level aspects of the compiler and run-time libraries. The following topics are most important:

- Program Sections
- Data Objects
- Assembly-Language Interfaces

The next subsections discuss each of these in turn.

5.1 Program Sections

The compiler isolates you from the nasty details of the 68000 instruction set, but you still need to know a little bit about how your C program gets translated into executable code. In particular, some knowledge of where the compiler puts things is quite helpful when you're using overlays, interfacing assembly-language routines, or adapting the compiler output for special applications.

The object modules produced by the compiler consist of several sections, as follows:

> **Code Section**
> The code section contains the machine instructions that carry out your program operations. The Lattice compiler produces no self-modifying code sequences, so this section can be placed into read-only memory (ROM).
>
> **Initialized Data Section**
> This section contains all initialized data items, including literals and constants. It cannot be placed into ROM because it contains writable items.
>
> **Uninitialized Data Section**
> This section contains all uninitialized data items. It is cleared to zero by the startup routine and, of course, cannot be placed in ROM because all of its items are writeable.

Normally these three sections are unnamed, but you can give them names via the -s option on the LC command.

There are two other program sections, the stack and the heap, which are not part of the executable file but which form an important part of the final executing program.

5.1.1 Stack Area

The stack is a writeable memory area whose size is established by the AmigaDOS STACK command, with the default being four kilobytes. This area is

used during function calls for saving registers and passing arguments. Within a function, automatic variables are allocated from the stack.

Many C programs will run just fine with a four-kilobyte stack, while others require much more space.

5.1.2 Heap Area

The heap is a writeable area whose size is determined by the dynamic memory needs of the program. The library includes functions such as *malloc* and *free* which the program uses to obtain and release blocks from this area. If the heap is not large enough to handle a request, the library function calls upon AmigaDOS to provide more memory. And so, the heap size is limited by the amount of memory installed on the system and the amount that is currently being used.

5.1.3 Section Addressing

The compiler provides several options which allow you to fine-tune the addressing of the various program sections. The default compiler options were selected for "typical" applications, but you may need to override these to get the best results in your situation.

5.1.3.1 Base-Relative Addressing

The **-b** option directs the compiler to generate instructions containing 16-bit offsets when making direct references to the initialized and uninitialized data sections. This is called *base-relative addressing* because the offsets are relative to a *base address* that is kept in register A4. The alternative is called *absolute addressing*, because the data reference instructions contain 32-bit absolute addresses.

The typical base-relative load or store instruction requires only 4 bytes instead of the 6 bytes necessary for absolute addressing. This can produce a substantial size reduction and performance improvement in a program that freqently accesses small data objects.

When you specify the **-b** option on an LC command, the compiler places information in the object module directing the linker to merge the data sections into a common area, which is called the *default data segment*. A linker error occurs if this segment is larger than 64 kilobytes, because that would prevent addressability via a 16-bit offset. At run time, the start-up routine loads A4 with the address of the default data segment.

If a program compiled with **-b** is used as an interrupt handler or with the *AddTask()* function, you must also specify the **-y** option. The compiler then generates code that automatically loads the A4 register at the beginning of each function. This action is necessary because interrupt and task functions are not entered via the normal C start-up routine. Alternatively, you can declare the interrupt or task function with the *__SAVEDS* keyword, which causes that function to be compiled with the prologue that saves A4 and then sets it to the default data segment address.

It is possible to mix modules compiled with and without the **-b** option, as long as the modules compiled with **-b** do not reference any data in the modules compiled without it. If this rule is violated, then you must use the **smalldata** option when linking, which forces all data items except those defined with the *chip* keyword to be merged into the default data segment.

It is also possible to mix base-relative and absolute addressing in the same module via the *near* and *far* keywords, which are described later. To summarize this feature, an object declared as *near* is always accessed via base-relative addressing, while one declared as *far* is accessed via absolute addressing.

5.1.4 PC-Relative Branches

The compiler defaults to call external functions via pc relative addresses. However, the **-r** option causes function calls to be PC-relative, where PC stands for "program counter," not "personal computer."

As with base-relative data addressing, PC-relative program addressing uses instructions containing 16-bit offsets instead of 32-bit absolute addresses. This saves two bytes per function call and improves performance.

The 16-bit offset is a signed value that is added to the current 32-bit address

in the program counter. This implies that a function being called via PC-relative addressing must reside no more than 32 kilobytes above or below each point that calls it. If this is not the case, the linker automatically constructs an absolute branch instruction within the 32-kilobyte range and routes the call through that branch.

5.2 Data Objects

Most of the time you can take the various C data types for granted. That is, you simply define and use the appropriate data objects without worrying about their representations on the Motorola 68000 or about how the object program accesses them. However, a better understanding of this topic is important if you plan to link your C programs with assembly language or if you need to know the valid ranges for the various data types.

The following subsections contain a detailed discussion of C data types in the AmigaDOS environment.

5.2.1 Simple Data Types

The compiler allows you to create objects using five simple data types, also called arithmetic types:

KEYWORD	DESCRIPTION
char	character
short short int	short integer
long long int	long integer
float	single-precision floating point
double long float	double-precision floating point

If you specify a data object as *int* without *short* or *long*, then it is treated as a long integer.

The compiler maintains fundamental data objects in forms that can be efficiently manipulated by the Motorola 68000. Specifically, multi-byte data is stored with the high order byte in the lowest address, and floating point numbers are kept in either the IEEE or the FFP format. The following table gives the sizes of these objects:

TYPE	BYTES	MIN	MAX
signed char	1	-128	+127
unsigned char	1	0	255
signed short	2	-32,768	+32,767
unsigned short	2	0	65,535
signed long	4	-2,147,483,648	+2,147,483,647
unsigned long	4	0	4,294,967,295
float (IEEE)	4	+/-10E-37	+/-10E+38
double (IEEE)	8	+/-10E-307	+/-10E+308
float (FFP)	4	?	?
double (FFP)	4	?	?

Here are some points to remember about these fundamental types in the AmigaDOS environment:

1. All arithmetic objects are signed by default (i.e. can take on values less than zero). However, if you specify the **-cu** option, the compiler will treat all *char* objects as unsigned.

2. You can override the default signing rule by using the *signed* or *unsigned* keyword as a modifier in front of any integral type (i.e. *char, int, short, long*). Declaring an object as *signed* is usually redundant, since integral objects are signed by default, except in the one case noted above. However, judicious use of this keyword can enhance your program's portability, because some compilers may make certain arithmetic types unsigned by default. Also, it's a good idea to use the *unsigned* modifier for any object (e.g. a loop counter) that can never be negative, since the compiler may be able to generate better code if it knows this fact.

3. The natural size of an integer in the AmigaDOS environment is 32 bits. That is, the types *int* and *long* are equivalent. However, if you want your program to be portable to other environments, use *short* when you want a 16-bit integer, *long* when you want a 32-bit integer, and use *int* when either size will suffice. In most C environments, *int* defaults to the most efficient size.

4. In expressions involving several data types, the compiler generally carries out the computation in the "widest" data type that is involved. The conversion rules are:

EXPRESSION CONTAINS...	COMPUTATIONAL WIDTH IS...
char	char
short	short
long	long
float	double
double	double

   So, for example, if an expression contains items having types *char* and *long*, the *char* types are converted to *long* during the computation.

5. All floating point computations are done in double precision, which is the traditional C standard. This implies that your program will generally execute slower if you use the *float* data type, because of the conversions between the *float* and the *double* form. Many programmers mistakenly believe that *float* data is handled faster than *double.*

5.2.2 Complex Data Types

Complex data types, also called aggregates, can be built from the simple data types. First we'll review the three complex data type categories, and then we'll describe how they are stored in memory.

ARRAY
An array is a named collection of similar objects, and you refer to each object by a combination of the array name and an index value, which

can be a short or long integer. The first object in an array has index value 0, and the compiler automatically scales the other index values to allow for the size of the objects in the array. That is, the second object has index value 1 regardless of its type. Here's an example of several array declarations:

```
char buffer[80];
int values[10][20];
```

Note that an array can be multi-dimensional. Also, arrays have no theoretical size limitation, although the size is, in practice, limited by the storage class and access method, which will be discussed later.

STRUCTURE
A structure is a named collection of dissimilar objects, each of which also has a name. You refer to each object by a combination of the structure name and the object name. Here's an example of a structure declaration:

```
struct emp
  {
  char name[32];
  char address[32];
  char city[16];
  char state[2];
  char zip[12];
  int payrate;
  long empdate;
  } employee;

struct emp empx,*empy;
```

The first declaration defines a structure type *emp* and allocates space for an object of that type named *employee*. Then, the second declaration uses the *emp* type to allocate space for an object named *empx* and for a pointer named *empy* which points to such a structure. Here are some examples of how you access individual items in these various instances of this structure:

```
employee.payrate = 100;

empx.payrate = employee.payrate;

empy->payrate = empx.payrate;
```

In short, you use a period (.) between the name of the structure object and the name of the item, or you use an arrow (->) between the name of the structure pointer and the name of the item. This will become clearer when we discuss pointers and data storage classes in a little while.

UNION
A union is a named collection of objects which are aliases for the same memory area. For example,

```
union ints
  {
  short si;
  long li;
  } anyint;
```

defines a union type *ints* and allocates space for an object of that type named *anyint.* The object contains two elements *si* and *li,* which are a short integer and a long integer, respectively. These objects actually occupy the same space in memory. The items within a union are accessed exactly like those within a structure, using the period and arrow as described above, and the same size limitation applies to unions..

Aggregates can contain any type of object, including other aggregates. The compiler maintains a separate "name space" for each structure and union, and so there is no requirement to use globally unique names for the internal items.

5.2.3 Data Pointers

In addition to the simple and complex data types, Lattice C supports a data type known as a *pointer.* Pointers are simply objects that hold addresses of

other objects; that is, a pointer refers to a simple or complex data object, or to another pointer. In the AmigaDOS environment, pointers are always 32 bits wide.

5.2.4 Data Storage Classes

Each data object, whether simple or complex, has a "storage class" attribute which is either explicitly declared via various keywords or is determined by the context in which the declaration occurs. Proper use of the storage class can have a major impact on a program's size and performance.

A data object's storage class is one of the following: internal, external, automatic, formal, or register. These are described below.

Internal
An object is internal if the *static* keyword is present in its declaration or if it is declared outside of any function without an explicit storage class specifier. In the former case, the object is also private, since its definition is not exported to the linker in the object module. In the latter case, the object is public and can be accessed by other modules which declare it as an external object.

Internal objects are often called "static objects", but we prefer to avoid that designation because not all internals are declared with the *static* keyword. It is better to think of internal objects as those which actually allocate space within the object module. That is, an internal object does not require further information from any other object module in order to be completely specified at link time.

External
An object is classified as external if the *extern* keyword is present in its declaration and if it is not later declared in the same module outside the body of any function without the *extern* keyword. Storage is not allocated in the object module for an external item. Instead, the compiler places information in the object module that allows the linker to complete the definition by finding the object module that defines the object internally.

Automatic
An object is classified as automatic if the *auto* keyword is present in its declaration or if it is declared inside the body of any function without an explicit storage class specifier. It is illegal to declare an *auto* object outside the body of a function. Storage is allocated for this type of object on the stack during the execution of the function in which it is defined.

Formal
An object is classified as formal if it is a parameter (also called an "argument") to a function. Storage is allocated for formal items on the stack by the program that calls the function.

Register
If you declare an object with the *register* keyword, it will be placed in one of the machine registers if possible. Otherwise, it is treated as an internal or formal object, depending on the context of the declaration.

The compiler provides the **-x** option to change the storage class of implicit internals to external. That is, with the **-x** option active, any internal object that does not have the *static* keyword is defined as if it had the *extern* keyword. This allows a single header file to be used for all external data definitions. When the main function is compiled, the **-x** option is not used, and so the various objects are defined and made externally visible. When the other functions are compiled, the **-x** option causes the same declarations to be interpreted as references to objects defined elsewhere.

5.2.5 Data Access Method

Each data object, whether fundamental or complex, has an "access method" attribute which is either explicitly declared via various keywords or is determined by the context in which the declaration occurs. As with the storage class, the access method can have a major impact on a program's size and performance.

A data object's access method is one of the following: *chip*, *const*, *far*, *huge*, *near*, or *volatile*. These are described below.

chip

The *chip* keyword specifies an object that resides within the first 512 kilobytes of main memory. The linker will automatically take care of this for you.

const

The compiler uses the "constant" access method for objects declared with the *const* keyword and for literals defined in a compilation using the **-gc** option. For example,

```
char const x[] = "abc";
char far *p;

p = "pdq";
```

would produce two constant items. The first is named **x** and has the value "abc". The second has the value "pdq", but it has no name because it is a literal.

far

The compiler uses the "far" access method for objects declared using the *far* keyword. For example,

```
int far x,y,z;
```

declares three far integers. The **-r0** option on LC command causes all data declarations without a specific access method to be treated as *far*.

huge

The *huge* keyword is identical to *far* in the AmigaDOS environment. It is included for compatibility with other environments which use Intel processors instead of the Motorola 68000 family.

near

The compiler uses the "near" access method for objects declared using the *near* keyword. For example,

```
int near x,y,z;
```

declares three near integers. These are placed into the data section in such a way that they can be accessed via 16-bit offsets from the data section pointer in register A4. The **-b1** option on the LC command causes all data declarations without a specific access method to be treated as *near*. This is the default setting for the **-b** option. In other words, LC normally generates *near* objects in order to reduce program size and improve performance.

Notice that pointers to *near* objects are always 32 bits wide. The only time that the 16-bit access occurs is when the offset can be embedded within an instruction. For most *near* objects, this is frequently the case, and so the size and performance improvements can be substantial. However, if you normally address an object via a pointer, you will gain little by declaring that object as *near*.

volatile
The *volatile* keyword describes a data object that can be changed by forces outside the control of the declaring program. Examples of such objects are memory-mapped I/O registers and shared memory. When manipulating a *volatile* object, the compiler reads or writes the object whenever it is referenced. In other words, the compiler suppresses any optimizations that would tend to keep volatile objects in registers.

The *chip*, *far*, *huge*, and *near* keywords are extensions to the ANSI standard. The standard requires that each keyword extensions be preceded by a double underscore, such as *__near*. The Lattice compiler accepts the extended keywords in this form as well as in the more natural form without the double underscores.

5.2.6 Special Purpose Keywords

Several other special purpose keywords are available with the compiler. They are:

__regargs	This keyword defines a subroutine that is to be called with register parameters.

__stdargs This keyword defines a subroutine that is to be called with standard stack parameters.

__asm This defines a subroutine that takes its parameters in a specific register.

__SAVEDS This defines a subroutine that is to load up the global base pointer upon entry.

__interrupt This defines a subroutine that may be called from interrupt code.

5.2.7 Data Alignment

Even though the Motorola 68000 processor uses byte addresses, it requires that 16-bit and 32-bit words be aligned on even addresses. That is, the lowest bit of the address must be zero. Because of this requirement, the Lattice C Compiler inserts dummy bytes as necessary to achieve the correct alignment of integers, floats, doubles, and pointers.

This can cause problems when you move data to or from systems that do not require alignment. For example, consider the structure:

```
struct misc
  {
  char x[3];
  int y;
  };
```

When an instance of this structure is defined, the compiler ensures that it begins on an even address by inserting a dummy byte into the data section if necessary. It also places a dummy byte between *x* and *y* so that the latter will be aligned correctly.

Notice that dummy bytes are counted by the *sizeof* function, which implies that

```
sizeof(struct misc)
```

is not equal to

```
sizeof(misc.x) + sizeof(misc.y)
```

The first expression has a value of 6, while the second is 5. This discrepancy can cause subtle problems in some programs.

5.2.8 Data Portability

You must be very careful when exchanging data files between dissimilar systems or between programs that use different alignment rules on the same system. For example, consider this short program which simply writes a structure to a file:

```
#include <stdio.h>
main()
{
FILE *fp;
struct
  {
  char x;
  short y;
  } record;

fp = fopen("testfile","wb");
record.x = 3;
record.y = 4;
fwrite(&record,sizeof(record),1,fp);
fclose(fp);
}
```

If you compile and execute this program using the default LC options, the file *testfile* will contain four bytes in the following order: 03 00 00 04. However, if you compile and execute this program under MS-DOS, which uses the Intel 8086 processor, the file will contain 03 04 00. Why? Because first of all, the Amiga compiler inserts a padding byte so that integer *y* is properly aligned; this is not necessary with the Intel processor. Second, the Motorola processor writes integers from the high byte to the low byte, while the Intel

processor does the reverse. This all goes to prove that portable programs do not necessarily produce portable data files.

If you are concerned about data file portability, you will save yourself a lot of grief by thinking carefully about the problem when designing file formats. The easiest way to handle it is to only write ASCII text files, since text file processing is usually fully portable. If you cannot do this, then you'll need to write a special translation program to re-align arrays and structures and to re-arrange the bytes in integers. These programs are known in UNIX circles as "data swabbers", and they are nasty to write and maintain.

One last comment on data portability: pointers are completely useless outside of the program that generates them, and so you should always avoid writing them to files. In fact, it is often true that pointers saved to a file are meaningless even when read back into the same computer system by the same program, because the data being pointed to will not necessarily be in the same place under different execution conditions.

5.3 Assembly-Language Interfaces

As mentioned earlier, it is common practice to develop a program entirely in C and then performance-tune it by re-writing certain functions in assembly language. Also, there are some operations, such as direct access of special CPU and I/O registers, that are not easily handled in C.

In this section we'll examine the entry and exit rules that must be followed by assembly language functions that are called from C functions.

5.3.1 Function Entry Rules

Upon entry to a function, the stack usually contains all the function arguments, which are placed immediately above the 4-byte return address. Register A7 is the stack pointer, and it points to the return address. The arguments appear in left-to-right order; that is, the leftmost argument is immediately above the return address in the stack. All arguments are passed by value.

For example, consider the function call:

```
char ccc;
double ddd;
int iii;

func(ccc,ddd,iii);
```

The compiler generates code to push the arguments *ccc*, *ddd*, and *iii* onto the stack, thereby using four bytes, eight bytes and four bytes, respectively. Next, *func* is called, which causes the four-byte return address to be pushed onto the stack.

The function must first perform the following entry sequence:

1. Save register A5, which contains the previous function's stack frame pointer.

2. Move A7 to A5, thereby establishing the frame pointer for this function. The arguments can then be addressed from A5 in the following way:

LOCATION	SIZE	CONTENTS
(A5)	4	Previous frame pointer
(A5)+4	4	Return address
(A5)+8	4	Argument *ccc*
(A5)+12	8	Argument *ddd*
(A5)+20	4	Argument *iii*

3. Subtract from A7 the size of the function's work area. These first three steps can be accomplished with one LINK instruction, if the work area is less than 32 kilobytes.

4. Save registers D2 through D7, A2 through A4, and A6 if they will be changed during execution of the function.

5. Save floating point registers FP2 through FP7 if a 68881 math coprocessor is present and will be used by the function.

6. If the function expects an argument to be a structure or union passed by value, then it will receive a pointer to that object in the argument list. Using the pointer, the object should be copied into the work area, and the work area copy should then be used within the function. Of course, it is not necessary to make a copy if the function only reads the object.

5.3.2 Register Arguments

If the calling function was compiled with the **-rr** option, then it passes some of the arguments in registers instead of on the stack. Specifically, the first two pointer arguments will appear in A0 and A1, and the first two integral arguments will be in D0 and D1.

Obviously, the function needs to know whether it is being called with some arguments in registers or with all arguments on the stack. The compiler helps make this distinction by placing the character @ in front of function names that are called with register arguments. This character replaces the underscore that the compiler normally supplies as a function prefix.

5.4 Function Exit Rules

Function return values are passed back in one or more registers, depending on the data type declared for the function. The conventions are:

RETURN DATA	BITS	ASM SYNTAX	MEANING
char	8	D0.B	Low byte of D0
short	16	D0.W	Low word of D0
long	32	D0.L	All of D0
float	32	D0.L	All of D0
pointer	32	D0.L	All of D0
double (IEEE)	64	D0.L,D1.L	High bits in D0
double (FFP)	32	D0.L	All of D0

If the function returns a structure or union, it must define a static work area (i.e. not on the stack) to temporarily hold the returned object. Then the function must return in D0 a pointer to this temporary copy, and the calling function will immediately move the data to the appropriate place. This approach implies that functions returning structures or unions are not re-entrant, although they are serially reusable. Such a function can be recursive if it is designed very carefully with this return technique in mind.

After setting up the return value, the function exits by reversing the entry sequence described earlier. Specifically, it must perform the following steps:

1. Restore registers that were saved.
2. Advance the stack pointer in A7 past the work area.
3. Restore the previous frame pointer to A5. This step and the preceding one can be accomplished with an UNLK instruction.
4. Return to the caller via an RTS instruction.

Note that the calling function removes the arguments from the stack.

Section 6

Macro Assembler

The Lattice Macro Assembler supports the development of assembly language modules for use with C programs. Because the Lattice C Compiler generally produces very good machine code, you seldom have to resort to assembly language programming. However, some intimate relations between hardware and software are best done in the assembly language environment. Also, assembly language is sometimes necessary when you want to get the absolute best combination of code size and speed.

The assembler handles the complete set of Motorola 68xxx instruction mnemonics as well as an extensive set of assembler directives and a powerful macro facility. It can, therefore, be used to develop complete systems in assembly language. Nonetheless, it is provided primarily to supplement the C compiler has not really been tuned for large assembly language projects.

6.1 Basic Concepts

The assembler reads a source file and produces an object file in the Amiga object file format, along with an optional listing of the source and assembled code. The source file is assumed to have a ".a" extension and the object file is produced with a ".o" extension. The BLINK linker, which is part of the

Lattice package, then combines these object modules into an executable file, also called a load module.

6.1.1 Source Format

Each assembly language source line has the following format:

```
label: operation operands ;comment
```

White space (i.e., spaces and horizontal tabs) can appear before any field and must appear between the operation and operand. The four fields of the source line are described below:

Label
The label field is optional. If it is present and is preceded by white space, it must be followed immediately by a colon. That is how the assembler determines that the field is a label and not an operation. If there is no white space before the label, then the colon is not necessary.

A label can be up to 31 characters long and can contain letters, digits, underscores, and dollar signs. It cannot start with a digit, and the case of letters is significant. For example, labels *XYZ*, *xYZ*, and *XyZ* are distinct.

Operation
The operation field contains the name of an instruction, assembly directive, or macro. This field may not begin a line. If no label is present, then the line must begin with white space. If a label is present but is not followed by a colon, then white space must separate the label and operation fields.

The case of this field is not significant. That is, operation *MOVE* is the same as *move*.

Operands
The operands field contains zero or more expressions, depending on the particular operation. For some operations, the operands field is

optional or never used. Expressions are composed of constants, variables, and operators.

A *constant* is a decimal, hexadecimal, octal, or binary number. The default number base is decimal, and the other bases are indicated by a suffix letter after the number:

Number Representations

Number	Representation	Example
Decimal	a string of decimal digits	1234
Hexadecimal	$ followed by a string of hex digits	$89AB
Octal	@ followed by a string of octal digits	@743
Binary	% followed by zeros and ones	%10110111
ASCII Literal	Up to 4 ASCII characters within quotes	"AC9T"

A *variable* is a label name or a name defined via an assembler directive.

An *operator* is one of the following:

ORDER	OPERATOR	MEANING
1	-	Unary minus
2	>>	Right shift
	<<	Left shift
3	*	Multiply
	/	Divide
4	+	Add
	-	Subtract
5	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
	==	Equal to
	!=	Not equal to
6	&	Bitwise AND
	!	Bitwise OR

The ORDER column indicates the order in which operators are processed. For example, in the expression

```
ABC+DEF*-PDQ
```

The negation of PDQ is performed first, followed by the multiplication and then the addition.

Each expression represents a 32-bit value. An *absolute expression* is one that contains only constants, while a *relocatable expression* contains symbols whose value is determined during the linking and loading procedure.

Comment
This field is any text preceded by a semicolon.

6.2 Using the Assembler

The assembler is invoked via the following CLI command:

```
asm [>listfile] [options] filename
```

Optional fields are enclosed in brackets, and all fields are described below:

> listfile
Causes the listing and error message output of the assembler to be directed to the specified file.

options
Assembler options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter. The options are:

-c
This option specifies which program sections should be loaded into memory that is addressable by the Amiga's custom hardware. The **-c** option must be immediately followed by one or more of the following letters in any order:

b Uninitialized (BSS) data section

c Code section

d Initialized data section

Without this option, the loader prefers to place all program sections into memory that is not also used by the custom hardware. This improves system performance by reducing bus contention. However, image and audio data must be loaded into memory that is accessible to the custom hardware. For example:

```
-abcd
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

-d
This option by itself causes debugging information to be placed into the object module. Another use of **-d** is to define symbols, as shown below:

-dABC
Causes ABC to be defined as if your source file began with the statement:

```
ABC EQU 1
```

-dABC=10
Causes ABC to be defined as if your source file began with the statement

```
ABC EQU 10
```

-h
This option is the reverse of **-a**. It specifies which program sections must not be placed into memory that can be addressed by the Amiga's custom hardware. As with **-a**, the **-h** option must be followed by one or more of these letters (-hbcd is the default):

b Uninitialized (BSS) data section

c Code section

d Initialized data section

-iprefix
This option specifies a prefix that should be placed in front of file names in INCLUDE statements. This is only done if the file name is not already prefixed by a drive or directory name. Up to four **-i** options can be used, and the INCLUDE search will use the prefix strings in that order. For example,

```
asm -iMYINC/ -iYOURINC/ testprog.a
```

assembles the source file named *testprog.a*. Suppose that source file contains the following statement:

```
INCLUDE ABC.I
```

The assembler first searches the current directory for the file named *ABC.I*. If that fails, it searches for *MYINC/ABC.I* and then *YOURINC/ABC.I*.

-l[list]
This option causes a listing of the source file to be written to the standard output. The listing displays the location counter and code or data alongside the assembly source. One or more of the following characters may be appended to the -l option:

- **i** Lists the source for text from INCLUDE files as well as the original source file.
- **m** Lists additional data generated for source lines which cannot be accommodated alongside the original source line (i.e., allows multiple listing lines for each source line).
- **x** Lists the expansion text for macros.

-m0
This option indicates that only 68000 instructions are allowed. Warning messages occur if, for example, you use instructions that are only available on the 68020.

-m2
This option indicates that 68020 instructions are allowed.

-oprefix
This option specifies a prefix for the output file name, that is, for the name of the ".o" file. Any drive or directory prefixes originally attached to the input filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the **-o**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

-s
This option includes the section name at the beginning of each hunk.

-u
This option prefixes external names with an underscore (_), which is useful when calling functions that were produced by the C compiler. Do not use this option if you type the external names with leading underscores.

Note that this option must be present when you assemble the start-up routine.

filename
Specifies the name of the source file which to be assembled. This is the only required field on the command line. If the name does not have an extension, ".a" is assumed. The object file will have the same name as the source file, except that the source file extension is replaced with ".o".

For example, the following command causes the assembly language source file *modn.a* to be assembled, producing the object file *modn.o*. A listing of the source file, along with any error messages generated, will be written to the file *modn.lst*.

```
asm >modn.lst -l modn
```

6.3 Assembler Directives

The assembler handles the all 68000, 68020, and 68030 instructions using the standard Motorola syntax. In addition, the ENTRY, PUBLIC, EXT, and EXTRN directives are processed in the standard way. All assembler directives are instructions to the assembler rather than instructions to be translated into object code. An exception to this is **DC**.

The following is a table of all of the assembler directives:

CNOP	Conditional NOP for alignment.
DC.B	Define Constant. Defines a constant value in memory. **.B** indicates that a character size is used or byte.
DC.L	Define constant. Defines a constant value in memory. **.L** indicates that a long size is used.

DC.W Define constant. Defines a constant value in memory. **.L** indicates that a word size is used.

DS.B Define Storage. Defines a byte of storage. DS does no initialization.

DS.L Define Storage. Defines a long word of storage. DS does no initialization.

DS.W Define Storage. Defines a word of storage. DS does no initialization.

DSECT Sets the location counter to the data segment. This is equivalent to the statement:

```
CSECT    data
```

ELSE Begins an alternative for IF directive.

END Program end.

ENDC End of conditional assembly.

ENDM End of macro definition.

EQU Assigns a permanent value.

EXITM Exit the macro expansion.

IDNT Name of program unit.

IF Assembles the following statements to the next ELSE or ENDIF if the expression supplied as its argument is not zero. IF statements may be nested.

IFC Assembles if the strings are identical.

IFD Assembles if symbol is defined.

IFEQ Assembles if expression is zero.

IFGE Assembles if expression is greater than or equal to zero.

IFGT Assembles if expression is greater than zero.

IFLE	Assembles if expression is less than or equal to zero.
IFLT	Assembles if expression is less than zero.
IFND	Assembles if symbol is not defined.
IFNE	Assembles if expression is not equal to zero.
INCLUDE	Insert file into source.
LIST	Turn on listing.
MACRO	Define a macro name.
NOLIST	Turn off listing.
OFFSET	Defines offsets. To define a table of offsets via the DS directive, you use the OFFSET directive.
OPSYN	Defines synonym for opcodes.
PAG	Page throw into listing.
RORG	Defines relocatable origin.
SECTION	Defines a program section.
SET	Assign a temporary value.
SPC	Skip n blank lines.
TTL	Disable object code output.
XDEF	Define an external name.
XREF	Reference an external name.

Note that $ or other special characters are not necessary with the assembly directives. Also, as with instruction mnemonics, directives cannot begin in the first character of the input line.

6.4 Macro Definition

A macro is defined via the following sequence, where the brackets enclose optional fields:

```
     MACRO
[labelarg]  macroname [arglist]
      .
      .
      .
     ENDM
```

The definition must begin with a MACRO directive on a line by itself. The next line is a model showing how the macro will be called. This is followed by the lines that comprise the macro procedure, and the definition is terminated with an ENDM directive on a line by itself. The directive EXITM can be used within the macro procedure to terminate processing early, possibly as a result of an IF test.

Labelarg, if present, is an identifier that can be used within the macro procedure to obtain the label. *Arglist* is a comma-separated list of argument string in the following format:

```
argsym[=default]
```

where *argsym* is an identifier that can be used within the expansion text to obtain the corresponding argument text used on the macro invocation line. *Default* is a character string that will be associated with *argsym* when that argument is not used in a particular macro invocation. Note that *default* must be enclosed in single or double quotes if it contains any whitespace characters.

Here is an example of a simple macro:

```
      MACRO
LABEL:   ACOPY  DEST1,A1
LABEL:   MOVE.W   4(SP),DEST1
         MOVE.L   6(SP),A1
      ENDM
```

The macro name is ACOPY, and it could be invoked in the following way:

```
ACOPY   d0,a5
```

resulting in the expanded text

```
MOVE.W  4(SP),d0
MOVE.L  6(sp),a5
```

A label argument, such as LABEL in the preceding example, must be specified and placed on the first expansion line of the macro if you want to be able to assign a label to the code generated by the macro.

6.5 Other Information

Of course, you cannot use the Lattice Macro Assembler unless you thoroughly understand the Motorola 68xxx instruction set. There are many books on this topic, some of which are cited in Appendix A.

Also, if you are writing assembly language functions to be called from C, or if you will call C functions from your assembly language programs, then you need to understand the Lattice C function protocol. This is described in Section 5 of this *User's Guide.*

Section 7

Global Optimization

7.1 Introduction

The global optimizer *GO* analyzes a quad file, performs several types of optimizations, and produces another quad file. This type of transformation makes the use of the optimizer completely optional since its input file is the same format as its output file. In many cases, optimized code is more difficult to debug than non-optimized code so frequently the optimizer is only used after the main program has been tested and is mostly working.

Since the optimizer works on quads, it has no knowledge of the target processor or its instructions. The code generator contains all of this knowledge and makes very full usage of the 680x0 instruction set. The code generator tries not to generate extra instructions in the first place but it does have a peep-hole optimizer to catch the few places where this is not possible.

7.2 Types of optimizations performed

1. **Register assignment**

   Commonly used auto, formal, and temporary variables are assigned to registers for all or part of their lifetime, according to usage.

2. **Dead store elimination**

   Stores of values which are never fetched again are eliminated.

3. **Dead code elimination**

   Code whose value is not used is eliminated.

4. **Global common subexpression merging**

   Recalculation of values that have been previously computed is eliminated. *GO* performs this with function scope.

5. **Hoisting of invariants out of loops**

   Calculations performed inside a loop whose value is the same on each iteration of the loop is moved outside the loop.

6. **Induction variable transformations**

   Loops containing multiplications, usually associated with indexing, have the operations reduced in strength to addition.

7. **Copy propagation**

   Definitions of the form leftvar = rightvar are eliminated when all uses of leftvar have this definition as the single reaching definition, and the variable rightvar will not change before each use. This optimization primarily exists to support other optimizations.

8. **Constant propagation and folding**

   References to variables whose only definition is a constant are replaced by the constant. Often the definition is eliminated if all references are replaced. *GO* performs constant folding to propagate the new constants further.

9. **Auto variable elimination and remapping**

   Unused auto variables are eliminated, and storage offsets are reassigned. Often the variable is unused because of previous optimizations.

10. **Very busy expression hoisting**

Code size is reduced by moving an expression computed along all paths from a point in the code to a common location. For instance, in

```
if (a())
   f(i + j);
else
   g(i + j);
```

the expression i+j will be computed in only one place.

11. **Various reductions in strength**

    GO will perform associative reordering of additive operations involving constants to reduce the operation count.

    Various arithmetic operations involving constants are reduced in strength.

    Conditional and logical expressions whose result is unused are converted into corresponding if( ) code. For instance, putchar( ) from <stdio.h> is implemented with a conditional expression. If the result (the original character or an error indication) is not used, *GO* converts it into if-else code, eliminating a load into a register.

12. **Various control flow transformations**

    GO will perform various transformations to eliminate unreachable code or useless control structures.

13. **Reordering of operations to reduce value lifetimes**

    Expressions with a single use are moved adjacent to the operation that uses them. This helps reduce temporary lifetimes, and supports optimizations that move code around. For example, in

    ```
    p[i] = f( );
    ```

    the computation of the address &p[i] can be moved after the call.

Appendix A

References

A.1 C Language References

Allen, Boris [1986]
Introducing C, ISBN 0-00383-105-1, William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, England.

Bolsky, M.I. [1985]
The C Programmer's Handbook, ISBN 0-13-110073-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Brand, Kim Jon [1985]
Common C Functions, ISBN 0-88022-069-4, QUE Corporation, Indianapolis, IN 46250.

Gehani, Narain [1985]
Advanced C: Food For The Educated Palate, ISBN 0-88175-078-6, Computer Science Press, Rockville, MD 20850.

Harbison, Samuel P. and Guy L. Steele Jr [1984]
A C Reference Manual, ISBN 0-13-110008-4, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Hogan, Thom [1984]
The C Programmer's Handbook, ISBN 0-89303-365-0, Brady Communications Company, Inc., Bowie, MD 20715.

Hunt, William James [1985]
The C Toolbox, ISBN 0-201-11111-X, Addison-Wesley Publishing Company, Reading, MA.

Jamsa, Kris [1985]
The C Library, ISBN 0-07-881110-4, McGraw-Hill, Berkeley, CA 94710.

Kernighan, Brian W. and Dennis M. Ritchie [1978]
The C Programming Language, ISBN 0-13-110163-3, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Kernighan, Brian W. and Dennis M. Ritchie [1988]
The C Programming Language, Second Edition, ISBN 0-13-110370-9, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Lawrence, David and Mark England [1985]
The C Compendium, ISBN 0-946498-86-6, Sunshine Books, 12-13 Little Newport Street, London WC2H 7PP, England.

Plum, Thomas and Jim Brodie [1985]
Efficient C, ISBN 0-911537-05-8, Plum Hall, Cardiff, NJ 08232.

Plum, Thomas [1983]
Learning to Program in C, ISBN 0-911537-00-7, Plum Hall, Cardiff, NJ 08232.

Purdum, Jack [1983]
C Programming Guide, ISBN 0-88022-022-8, QUE Corporation, Indianapolis, IN 46250.

Purdum, Jack [1985]
C Self-Study Guide, ISBN 0-88022-149-6, QUE Corporation, Indianapolis, IN 46250, USA.

Radcliffe, Robert A. and Thomas J. Raab [1986]
Data Handling Utilities In C, ISBN 0-89588-304-X, Sybex, Berkeley, CA 94710.

Schildt, Herbert [1986]
Advanced C, ISBN 0-07-881208-9, McGraw-Hill, Berkeley, CA 94710.

Schustack, Steve [1985]
Variations In C, ISBN 0-914845-48-9, Microsoft Press, Bellevue, WA 98009.

Schwaderer, W. David [1985]
C Wizard's Programming Reference, ISBN 0-471-82641-3, Wiley Press, New York, NY 10158.

Sinclair, Ian [1986]
C For Beginners, ISBN 0-86161-206-X, Melbourne House Publishers Ltd., 60 High Street, Kingston-Upon-Thames, Surrey KT1 4DB, England.

Sobelman, Gerald E. and David E. Krekelberg [1985]
Advanced C Techniques & Applications, ISBN 0-88022-162-3, Que Corporation, Indianapolis, IN 46250.

Stevens, Al [1986]
C Development Tools For The IBM PC, ISBN 0-89303-612-9, Prentice Hall, New York, NY 10023.

Traister, Robert J. [1985]
Going from BASIC to C,ISBN 0-13-357799-6, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Traister, Robert J. [[1984]
Programming In C For The Microcomputer User, ISBN 0-13-729641-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Ward, Robert [1986]
Debugging C, IDBN 0-88022-261-1, QUE Corporation, Indianapolis, IN 46250, USA.

A.2 AmigaDOS References

Ariadne Software [1987]
The Kickstart Guide to the Amiga, Ariadne Software Ltd. 273 Kensal Road, London NW10, England.

Berry, John Thomas
Inside the Amiga, ISBN 0-672-22468-2, Howard W. Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

Boom, Michael [1986]
The Amiga: Images, Sounds and Animation ISBN 0-914845-62-4, Microsoft Press, Box 97017, Redmond, WA 98073-9717, USA.

Commodore-Amiga, Inc. [1986]
The AmigaDOS Manual, ISBN 0-553-34294-0, Bantam Electronic Publishing, 666 Fifth Avenue, New York, NY 10103, USA.

Commodore-Amiga, Inc. [1986]
Amiga ROM Kernal Reference Manual: Exec, ISBN 0-201-11099-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]
Amiga ROM Kernel Reference Manual: Libraries and Devices, ISBN 0-201-11078-4 Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]
Amiga Intuition Reference Manual, ISBN 0-201-11076-8, Addison-Wesley Publishing Company, Reading, MA, USA.

Commodore-Amiga, Inc. [1986]
Amiga Hardware Reference Manual, ISBN 0-201-11077-6, Addison-Wesley Publishing Company, Reading, MA, USA.

Compute! Publications, Inc. [1986]
Inside Amiga Graphics, ISBN 0-87455-040-8, Compute! Publications, Inc., P.O. Box 5406, Greensboro, NC 27403, USA.

Donald, Bill [1986]
The Amiga System: An Introduction, ISBN 1-85231-001-4, Precision Software Ltd. 6 Park Terrace, Worcester Park, Surrey KT4 7JZ, England.

Mortimore, Eugene P. [1986]
Amiga Programmer's Handbook, Volume I, ISBN 0-895880-343-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Mortimore, Eugene P. [1987]
Amiga Programmer's Handbook, Volume II, ISBN 0-895888-384-8, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Peck, Robert A. [1987]
Programmer's Guide To The Amiga, ISBN 0-895883-310-4, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Spanik, Christian [1986]
Presenting The Amiga, ISBN 0-948015-128, First Publishing Ltd., Horseshoe Road, Pangbourne, Berkshire RG8 7SW, England.

A.3 Motorola 68xxx References

Barrow, David [1985]
68000 Machine Code Programming, ISBN 0-00-383163-9, William Collins Sons & Co.Ltd., 8 Grafton Street, London W1X 3LA, England.

Eccles, William J. [1985]
Microprocessor Systems: A 16-Bit Approach, ISBN 0-201-11985-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Kane, G., D.Hawkins and L.Leventhal [1987]
68000 Assembly Language Programming 2nd Edition, ISBN 0-07-881232-1, Osborne/McGraw-Hill, 2600 Tenth Street, Berkely, CA 94710, USA.

Motorola Inc. [1986]
The MC68000 User's Manual 5th Edition, ISBN 0-13-541475-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Motorola Inc. [1985]
The MC68020 User's Manual 2nd Edition, ISBN 0-13-541475-X, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA.

Motorola Inc. [1985]
The MC68881 User's Manual, Motorola Semiconductor Products Inc., PO Box 20912 Phoenix, AZ 85036, USA.

Robinson, Phillip R. [1985]
Mastering The 68000 Microprocessor, ISBN 0-8306-1886-4, Tab Books Inc., Blue Ridge Summit, PA 17214, USA.

Kelly-Bootle, Stan and Bob Fowler [1985]
68000, 68010, 68020 Primer, ISBN 067-22405-4, Howard W.Sams & Co., 4300 W.62nd Street, Indianapolis, IN 46268, USA.

Williams, Steve [1985]
Programming the 68000, ISBN 0-89588-133-0, SYBEX Inc., 2021 Challenger Drive #100, Alameda, CA 94501, USA.

Whitworth, Ian R. [1985]
16-Bit Microprocessors, ISBN 0-00-383113-2, William Collins Sons & Co. Ltd., 8 Grafton Street, London W1X 3LA, England.

Appendix B

Diskette Contents

This appendix lists the contents of the three disks which comprise the Lattice Amiga C Compiler product.

- Disk 1 is a bootable system disk containing the most used executable programs.
- Disk 2 contains the Lattice and Amiga *header* files and libraries in compressed form.
- Disk 3 contains the remaining executables (CPR, GO, LC1B, utilities,...)
- Disk 4 contains examples, lseinst, uncompressed header files, some source files, and some library files (lcnb.lib, lcsnb.lib).

Contents of disks

Disk 1: Workbench stuff (just like 4.0)

Directory	File
root	readme
	PopCLI
s	
	install hd
	install floppy
	startup-sequence
c	
	lc
	lc1
	lc2
	lcerrs.txt
	lse
	lse.dat
	lse.msg
	lse.hlp
	blink
	oml

Disk 2: Libraries and Headers

Directory	File
lib	
	lc.lib
	lcm.lib
	lcmffp.lib
	lcmffps.lib
	lcmieee.lib
	lcms.lib
	lcs.lib
	lcrs.lib
	lcr.lib
	lcsnb.lib
	amiga.lib
	grep.lib
	c.o
	cres.o
	catch.o
	catchres.o
	cback.o
CompactH	
	all the compressed header files

Disk 3: Other Utilities and Executables

Directory	File
c	
	asm
	omd
	cxref
	diff
	files
	grep
	lmk
	touch
	wc
	build
	extract
	splat
	lstat
	lprof
	cpr
	cpr.hlp
	go
	dumpobj
	fd2pragma
	lcompact

Disk 4: Examples, Non-Compressed Headers, and Extras

Directory	File
include	
	all the non compressed header files
lib	
	lcnb.lib
	lcsnb.lib
source	
	cres.a
	c.a
	catch.a
	catchres.a
	cback.a
examples	
	all the examples
cc extras	
	grep.lmk
	grepdemo.c
	grepdemo.doc
	lmk.def
	lmkfile
	pat.h
	touch.doc
	wc.c
lseinst	
	lseinst
	lseinst.msg

Appendix C

Upgrade Guidelines

C.1 Purpose

This appendix is intended to help you upgrade to Version 5 from previous editions of the Lattice AmigaDOS C Compiler. As you'll see, this requires very little work.

C.2 Version 4 Upgrade

Version 5.0 is fully upward compatible with Version 4.0. Object modules and libraries created with either compiler may be freely mixed. Of course, some of the new features (e.g., register parameters and debugging support) are only available to code compiled with the Version 5 compiler.

Here is a list of the improvements and changes in Version 5, relative to Version 4:

1. The compiler now includes a full ANSI preprocessor with string facilities, token facilities and appropriate scoping of substitution symbols. The defined() directive is also supported. In addition, __DATE__ and __TIME__ provide the date and time of compilation.

2. The compiler now uses sequence points to ensure correct evaluation and side effect generation according to the draft standard.
3. The *const* and *volatile* keywords are supported.
4. Function prototypes may now include an optional parameter name. Also functions taking a variable number of arguments may be indicated with ellipses '...'.
5. String literals may now be concatenated to allow easier coding of long strings.
6. The cast operation *(void *)* correctly coerces a type without any warning.
7. Many diagnostic messages have been added to detect programs that do not conform to the ANSI standard.
8. The compiler now recognizes several new keywords:

signed	Overrides any default unsigned options.
near	Declares a data item to be addressed relative to the global base register. When used with a subroutine, it indicates a PC-relative subroutine call.
far	Declares an item that must be addressed with a full 32 bit address.
huge	Same as *far*.
chip	Declares a data item that must be placed in chip ram and addressed with a full 32 bit address.
__regargs	Defines a subroutine that is to be called with register parameters.
__stdargs	Defines a subroutine that is to be called with standard stack parameters.
__asm	Defines a subroutine that takes its parameters in a specific register

__SAVEDS Defines a subroutine that is to load up the global base pointer upon entry.

__interrupt Defines a subroutine that may be called from interrupt code.

9. To provide for faster compilation of a large project, the symbol table may be saved out to disk so that it can be used in future compilations. If you run a large header file through the compiler and save it in this way, subsequent compilations using that header file will be much quicker.

10. The compiler provides an option to ignore redundant *#include* statements (i.e., several *#include* statements that refer to the same file).

11. The search rules for *#include* files have been modified to conform to standard UNIX search conventions.

12. Using the **lstat** and **lprof** command, we have done a significant amount of fine tuning the compiler to provide optimum performance on the Amiga.

13. To allow more error messages to appear on the screen before scrolling off, we have adopted a more condensed format of displaying the error position in reverse video on the source line followed by the error message. This eliminates the visual confusion caused by the seemingly blank line between the message and the line. With the old format, there was a tendency to associate the error message with the wrong line.

14. It is now possible to disable particular error messages as well as to change the severity of most messages. Along with this, it is now possible to specify a maximum number of errors allowed in a compilation so that the compiler will abort.

15. We have implemented an improved form of error recovery for many of the common mistakes to eliminate many of the situations that resulted in a cascade of errors.

16. The compiler supports a limited form of structure comparision of EXACTLY identical structures which have no holes. This is implemented

as a call to memcmp (which is a "builtin" function) to perform a byte-by-byte comparison of the two structures.

17. The library base for *#pragma* statements is no longer limited to being a single external pointer. Any arbitrary expression may be used including function calls and local variables. This is indented to support generating code that resides in a library.

18. All of the compiler messages have been moved to a separate file to simplify adaptation of the compiler to environments outside of the USA.

19. In order to produce a compiler that fits well on a smaller system, we have elected to provide a "big version" of the compiler that includes some additional features. If you wish to take advantage of these features (at a cost of about 20K), you must use this big compiler instead of the standard one.

    - The big compiler provides a full listing ability including macro expansion display, nest level counting, and include file listing. This listing may also include an optional cross reference of all variables, #define values and structure tags.

    - The big version of the compiler may be used to generate prototype files of all functions encountered in a module. This eliminates the potentially tedious task of constructing the list of prototypes for all functions in a project.

20. The compiler generates instructions optimized for each of the 680x0 family processors including support for the address modes found on the 68020 and 68030.

21. The compiler provides an option to generate in-line floating point instructions which directly access the 68881 and 68882 math coprocessor. This code takes advantage of register tracking.

22. The compiler can generate code that is fully compatible with the requirement of an Amiga library routine. Specifically, A6 is removed from the compiler selection list and A4 is made available for code use. Note that this code is incompatible with the **-b1** option.

23. You may now instruct the compiler to choose code sequences optimized for space or for time.

24. In addition to the **-r0/-r1** flags, you may freely mix the style of subroutine calls with the *near* and *far* keywords. Those declared *near* will be referenced with the pc-relative addressing while *far* will use the full 32-bit addressing.

25. Data may be addressed much more freely with the *near*, *far*, and *chip* keywords. These control the type of addressing to be used for external data. Only those items declared *near* will be addressed as a 16-bit offset from the A4 register. All others will be accessed with a full 32-bit address.

26. When the **-cs** option is used, string constants will be placed in the code section. This option is beneficial when generating resident modules as well as with the -b0 option.

27. To support the debugger, the compiler can now generate several levels of debugging information including the old **LINE** style debug sections.

28. Two styles of register parameters are supported. The **-rr** option causes the compiler to automatically place up to four parameters in registers for subroutine calls. The *__asm* keyword may be used in conjunction with a register specification list to cause the compiler to pass parameters in a given register.

29. The compiler now tracks the condition codes affected by the generated code in a attempt to avoid generating unnecessary test instructions.

30. Stack cleanup on subroutine calls is delayed as long as possible to allow coalescing and even elimination of the cleanup across multiple calls.

31. The bitwise boolean operations generate better code for dealing with constant values.

32. Division by 2, 4, or 8 no longer generates a subroutine call. The compiler generates inline code to normalize and perform the calculation.

33. Bit shift operations have been rewritten completely. The compiler now generates optimal shift sequences for all constant values.

34. Register variables no longer generate spurious double copies.

35. The compiler attempts to place as many variables as possible in registers unless this feature is explicitly disabled.

36. The library bases for the *#pragma* library calls are tracked to eliminate unnecessary reloading. If necessary, the compiler will swap library bases with another register.

37. The code generator now takes full advantage of ALL 68000 address modes including auto increment and PC-relative indexed. Tracking of indexing operations allows the compiler to suppress unnecessary adds and substitute indexed address modes.

38. Loading of specific constants has been optimized to generate the optimal code sequence and avoid **MOVE.L #** as much as possible.

39. Several new built-in functions have been added:

abs	Returns the absolute value of an integer.
max	Returns the larger of two integers.
min	Returns the smaller of two integers.
geta4	Establishes addressability of the global data register.
putreg	Directly stores into a specific 68000 register.
getreg	Obtains the contents of a specific 68000 register.
emit	Inserts a hex word into the instruction stream.

40. Many small code optimizations suggested by users have been implemented. In particular, the compiler no longer will generate NOP instructions. Also, MOVEM instructions in the prologue/epilogue that reference a single register are converted to MOVE instructions.

41. Switch statements on values which are in the range of a short are converted to use the more efficient code.

42. Support for the 68020 and 68030 processor and the 68881 and 68882 math co-processor have been added to the compiler, assembler, and other utility programs.

43. The assembler supports line number tables.

44. Many of the library routines have been recoded to take advantage of better algorithms. The most important routines were recoded in assembly language.

While we have worked hard to bring the compiler into ANSI compliance, there are still a couple of areas where we have not reached full compliance:

- Not all of the tri-graph sequences intended for support of incomplete keyboards are implemented.
- Some obscure keyword orderings in declarations are not correctly parsed.

Appendix D

Compiler Error Messages

D.1 Overview

The Lattice C Compiler produces three types of error messages:

Operational Errors
These indicate that the compiler is having trouble operating correctly because it cannot access required files or cannot obtain enough disk or memory space.

Syntax Errors and Warnings
These indicate that the compiler is having difficulty understanding your C Source program. The message includes the source file name and line number identifying the point at which the problem was detected. An "error message" indicates that the problem prevents the construction of a usable object module and must, therefore, be corrected. A "warning message" indicates that the compiler detected something unusual but will proceed to make an object module, using appropriate assumptions about what you intended the source code to do.

Internal Errors
These indicate that the compiler encountered some internal condition that should not have occurred. This is the old "I shouldn't be here" type of message, and you should report it to our Technical Support Department. However, before doing so, we would be grateful if you would conduct a few experiments with your source code to see if you can make the problem go away. The internal error explanations given below should provide enough clues.

D.2 Operational Errors

Can't create object file

The second phase of the compiler was unable to create the object file. This error usually results from a full directory on the output disk.

Can't create quad file

The first phase of the compiler was unable to create the quad file. This error usually results from a full directory on the output disk.

Can't open file for pre-processor output

The first phase of the compiler was unable to open the pre-processor output file. This error usually results from a full directory on the output disk.

Can't open prototype file

The first phase of the compiler was unable to open the prototype file. This error usually results from a full directory on the output disk.

Can't open quad file

The second phase of the compiler was unable to open the quad file. This error usually occurs when you call phase 2 of the compiler (*lc2.exe*) directly with an invalid quad file name.

Can't open source file

The first phase of the compiler was unable to open the source file. This error usually occurs because you misspelled the file name or did not specify the proper drive and/or directory path.

Can't open symbol file

The first phase of the compiler was unable to open the symbol file. This error usually results from a full directory on the output disk.

Combined output file name too large

The output file name constructed by combining the source or quad file name with the text specified using the **-o** option exceeded the maximum file name size of 64 bytes.

End of file on object file

The second phase of the compiler detected an end of file condition on the object file. This usually indicates a full disk.

Error reading symbol file

The second phase encountered an error when reading the debugging symbol file.

File name missing

The source file name was not specified.

File name too large

The name of the file passed to the second phase exceeded the maximum file name length.

Intermediate file error

The first phase of the compiler encountered an error when writing to the quad file. This error usually results from an out-of-space condition on the output disk.

Invalid -e option

The character following the -e was not a 0, 1, or 2. This usually occurs because the line was mistyped. Retype the line and try again. See the *Command Reference* for valid command line options.

Invalid command line option

An invalid command line option was specified, and that option will be ignored. See the *Command Reference* for valid command line options.

Invalid symbol definition

The name attached to **-d** specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equal sign.

No functions or data defined

The compiler reached the end of the source file without finding any data or function definitions. One common cause of this error is to forget a comment terminator (*/) during the first function in the source file. This causes the compiler to gobble up your program as if it were a comment.

Not enough memory

This message is generated when either phase of the compiler uses up all the available working memory.

Parameters beyond file name ignored

Additional information was present on the command line beyond the name of the source file. A common source of this error is to place compiler options after the file name, which was required on earlier versions of Lattice C.

Seek error on object file

The second phase of the compiler detected a seek error on the object file. This usually indicates a full disk.

Symbol file corrupted

The second phase detected a corrupt symbol file. This rarely occurs and may be related to a full disk or disk integrity.

Unrecognized -c option

One of the characters following the **-c** option was not a recognized compiler control character. See the LC command in the *Command Reference* for a list of the valid compiler control options.

i option ignored

More than 16 **-i** option strings were specified. Only the first 16 are retained and used.

D.3 Syntax Errors and Warnings

Syntax errors and warnings are reported via a message with the following format:

```
fff nnn Error xxx: mmm
```

where the message components are:

fff

This is the name of the source file that was being processed when the error occurred.

nnn

This is the number of the source file line that was being scanned when the error occurred. Source file lines begin at 1, not 0.

xxx

This is the error number, as listed below.

mmm

This is the error message text.

Note that all of these fields are variable length.

All messages indicate "fatal errors" unless the message number listed below is followed by **(W)**. When a fatal error occurs, the compiler will not produce

a usable object module. The LC command alerts you to this condition by beeping and pausing, unless you use the **-C** option to force continuous compilation.

If the message number below is followed by **(W)**, then it is a warning. When such a message is displayed, the compiler will produce a usable object module by making reasonable assumptions about what you intended the source file to do. Nonetheless, it's a good idea to investigate these warnings, since the compiler's assumptions may disagree with your intentions.

1 This error is generated by a variety of conditions in connection with pre-processor commands, including specifying an unrecognized command, failure to include white space between command elements, or use of an illegal pre-processor symbol.

2 The end of an input file was encountered when the compiler expected more data. This may occur on an *#include* file or the original source file. In many cases, correction of a previous error will eliminate this one.

3 The file name specified on an *#include* command was not found.

4 An unrecognized element was encountered in the input file that could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file.

5 A pre-processor *#define* macro was used with the wrong number of arguments.

6 Expansion of a *#define* macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line.

7 The maximum extent of *#include* file nesting was exceeded; the compiler supports *#include* nesting to a maximum depth of 16.

8 A cast (type conversion) operator was incorrectly specified in an expression.

9 The named identifier was undefined in the context in which it appeared, that is, it had not been previously declared. This message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 An error was detected in the expression following the [character (presumably a subscript expression). This may occur if the expression in brackets was null (not present).

11 The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 The expression preceding the period (.) or arrow (->) structure reference operator was not recognizable by the compiler as a structure or pointer to a structure.

13 An identifier indicating the desired aggregate member was not found following the period (.) or arrow (->) operator.

14 The indicated identifier was not a member of the structure or union to which the period (.) or arrow (->) referred.

15 The identifier preceding the left parenthesis of a function call was not implicitly or explicitly declared as a function.

16 A function argument expression following the left parenthesis on a function call was invalid. This may occur if an argument expression was omitted.

17 During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was poorly specified or improperly used.

20 An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for constant expressions was present.

21 An identifier declared as a structure or union was encountered in an expression where aggregates are not permitted. Only the direct assignment and conditional operators may be used on aggregates, and explicit or implicit testing of aggregates as a whole is not permitted.

22 (W) An identifier declared as a structure or union appeared as a function argument without the preceding & operator. In Version 3 of Lattice C, aggregates may be passed by value, so this is a legal construction. The warning message is generated to alert you that earlier versions of Lattice C passed the address of the aggregrate in this case.

23 The conditional operator was used erroneously. This may occur if the ? operator was present but the : was not found when expected.

24 The context of the expression required an operand to be a pointer. This may occur if the expression following * did not evaluate to a pointer.

25 The context of the expression required an operand to be an lvalue. This may occur if the expression following & was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26 The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (W) In an assignment statement defining a value for a pointer variable, the expression on the right side of the = operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e., it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. Note that the same message may be a fatal error if generated for an initializer expression or in some situations involving mixed memory models.

31 The context of an expression required an operand to be integral, i.e., one of the integer types (char, int, short, unsigned, or long).

32 The expression specifying the type name for a cast (conversion) operation or a sizeof expression was invalid.

33 An attempt was made to attach an initializer expression to a structure, union, or array that was declared auto. Such initializations are expressly disallowed by the language.

34 The expression used to initialize an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression which did not evaluate to a constant. Some experimentation may be required in order to determine the exact cause of the error.

35 During processing of an initializer list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initializer expression list or if a structure member was improperly declared.

36 A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 The specified statement label was encountered more than once during processing of the current function.

38 In a body of compound statements, the number of opening left braces { and closing right braces } was not equal. This may occur if the compiler got "out of phase" due to a previous error.

39 One of the C language reserved words appeared in an invalid context (e.g., as a variable name). Note that the keyword *entry* is reserved although it is not implemented in the compiler.

40 A break statement was detected that was not within the scope of a while, do, for, or switch statement. This may occur due to an error in a preceding statement.

41 A case prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

42 The expression defining a case value did not evaluate to an int constant.

43 A case prefix was encountered which defined a constant value already used in a previous case prefix within the same switch statement.

44 A continue statement was detected that was not within the scope of a while, do, or for loop. This may occur due to an error in a preceding statement.

45 A default prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

46 A default prefix was encountered within the scope of a switch statement in which a preceding default prefix had already been encountered.

47 Following the body of a do statement, the while clause was expected but not found. This may occur due to an error within the body of the do statement.

48 The expression defining the looping condition in a while or do loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 An else keyword was detected that was not within the scope of a preceding if statement. This may occur due to an error in a preceding statement.

50 A statement label following the goto keyword was expected but not found.

51 The indicated identifier, which appeared in a goto statement as a statement label, was already defined as a variable within the scope of the current function.

52 The expression following the if keyword was null (not present).

53 The expression following the return keyword could not be legally converted to the type of the value returned by the function.

54 The expression defining the value for a switch statement did not define an int value or a value that could be legally converted to int.

55 (W) The statement defining the body of a switch statement did not contain at least one case prefix.

56 The compiler expected but did not find a colon (:). This error message may be generated if a case expression was improperly specified, or if the colon was simply omitted following a label or prefix to a statement.

57 The compiler expected but did not find a semi-colon (;). This error generally means that the compiler completed the processing of an expression but did not find a statement terminator. This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through white space to look for the semi-colon, the line number for this error message may be beyond the actual line where a semi-colon was needed.

58 A parenthesis required by the syntax of the current statement was expected but was not found (as in a while or for loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 In processing declarations, the compiler encountered a storage class invalid for that declaration context (such as auto or register for external objects). This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 The types of the aggregates involved in an assignment or conditional operation were not exactly the same. This error may also be generated for *enum* objects, which are treated as integers.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file has been processed if a pointer was declared with a tag that was never defined.

62 A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e., a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of improper declarations.

64 A declaration of the members of a structure or union did not contain at least one member name.

65 An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to "get out of phase" with respect to declarations in the source file.

66 The expression defining the size of a subscript in an array declaration did not evaluate to a positive int constant. This may also occur if a zero length was specified for an inner (i.e., not the leftmost) subscript of an array object.

67 A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return arrays and arrays of functions.

68 A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the * is forgotten on a structure pointer declaration, or if (due to some intertwining of structure definitions) the structure actually contains an instance of itself.

70 The formal parameter of a function was declared illegally as a function.

71 A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit *int* function, and was then declared as returning some other kind of value. Functions which return a type other than *int* must be declared before they are used so that the compiler is aware of the type of the function value.

73 In processing the declaration of objects, the compiler expected to find another line of declarations but did not, in fact, find one. This error may be generated if a preceding error caused the compiler to "get out of phase" with respect to declarations.

74 (W) A string constant used as an initializer for a char array defined more characters than the specified array length. Only as many characters as are needed to define the entire array are taken from the first characters of the string constant.

75 An attempt was made to apply the sizeof operator to a bit field, which is illegal.

76 The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initializer expressions for an aggregate.

77 In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 The indicated statement label was referred to in the most recent function in a goto statement, but no definition of the label was found in that function.

79 (W) More than one identifier within the list for an enumeration type had the same value. While this is not technically an error, it is usually of questionable value.

80 The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of a machine word (such as 16 on a 16-bit machine); these must be declared as ordinary int or unsigned variables.

81 The current line contains a reference to a pre-processor symbol that is a circular definition.

82 The size of an object exceeded the maximum legal size for objects in its storage class; or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum.

83 (W) An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an expression makes reference to an element beyond the end of an array.

84 (W) A *#define* statement was encountered for an already defined symbol. The first definition is "pushed", so that an additional *#undef* statement is needed to undefine the symbol.

85 (W) The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type. This warning is also issued when a return statement with a null expression (i.e., no return value) appears in a function which was not declared void; generation of the warning for this particular context can be disabled using the **-cw** option on the LC command.

86 (W) The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of that function with argument type specifiers.

87 (W) The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (W) The type of a function argument expression did not agree with its corresponding type declared in the list of argument type specifiers for that function. Note that the compiler does not automatically convert the expression to the specified type; it merely issues this warning.

89 (W) The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function. In this case, the compiler does convert the expression to the expected type.

90 The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in cast operators or sizeof expressions.

91 One of the operands in an expression was of type void; this is expressly disallowed, since void represents no value.

92 (W) An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful

purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (W) An object with local scope was declared but never referenced within that scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (if, for example, the code in which the variable was used was eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives such as *#ifdef* or *#if*.

94 (W) An auto variable was used in an expression without having been previously initialized by an assignment statement or appearing in a function argument list with a preceding & (i.e., its address passed to a function). Note that the compiler considers the variable initialized once any statement causes it to be initialized, even though control may not flow from that statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a for statement uses a variable which has not yet been initialized, which may be incorrect if that variable is initialized inside the body of the for statement.

95 The program is writing to a *const* object.

D.4 Internal Errors

These errors are reported via the message:

```
CXERR: xx
```

where *xx* is the error number. When such a message occurs, compilation is terminated immediately, and both the quad file and the object file are probably unusable. In such a case, please note the message number and contact Lattice Technical Support for assistance.

1 Invalid error or warning message code number.

2	The compiler has internally called a function that is not applicable to the current host-target environment.
3	Invalid symbol table access.
4	Declaration chain is broken.
5	An unlink error occurred while processing an "undef" command.
6	The compiler attempted to push back too many tokens.
7	There is no aggregate list for a structure reference.
8	Stack underflow has occurred.
9	Invalid attempt to generate the address of a constant.
10	A test value is not a constant.
11	Invalid unary operator.
12	Invalid binary operator.
13	A scaling object is not a pointer or array.
14	An unexpected end-of-chain occurred while restoring internal context.
15	Invalid quad type.
16	Deletion length is less than two bytes.
17	Insufficient memory.
18	An error occurred when releasing memory.
19	Invalid condition during temporary assignment.
20	Invalid condition while processing program section.
21	Literal pool generation error.
22	Invalid condition while processing data section.
23	Invalid quad file.
24	End-of-file while processing "for" quad.

25	Invalid register number.
26	Temporary save or restore error.
27	Invalid operand size.
28	Invalid storage base.
29	Error during branch folding.
30	Error during control statement processing.
31	Error during special addressing setup.
32	Invalid object description block offset.
33	Too many function parameters.
34	Indirect argument for call-by-reference.
35	Invalid external relocation value.
36	Error during search of the debugging information lists.
37	Error during search of library lists.

Index

A

B

C

D

P

Q

R

S

T

U

Utilities

Utilities

SAS/C® Programmer Utilities Manual

Version 5.10

Utilities for Efficient Amiga® Programming

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C® Programmer Utilities Manual

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Lattice Programmer Utilities

Version 1.00

Utilities for Efficient Amiga Programming

Lattice, Inc.
2500 S. Highland Avenue
Lombard, IL 60148
USA

A Subsidiary of SAS Institute Inc.

Lattice Programmer Utilities Manual

This manual was formatted using **HighStyle**® by Lattice, Inc.

Table of Contents

APPENDICES

Section 1

Overview

The Lattice Programmer Utilities offers professional programmers a collection of utilities that provide greater control and functionality over the programming environment. Each utility operates as an independent task within the multi-tasking operating system of the Commodore Amiga. This means that editing or compilation cycles can continue uninterrupted while the programmer initiates further operations using one of the utilities. The ability to take advantage of multi-tasking in this way as well as the range of facilities provided by this package compliments and enhances the **Lattice Amiga C Compiler**.

The Utilities will operate on PAL or NTSC versions of the Commodore Amiga 500 and 1000, as well as both variants of the 2000 model.

1.1 Description of Tools

The Lattice Utilities contains a total of thirteen utilities for use by programmers. The purpose of this part of the Lattice Amiga C Manual is to serve as a reference source for each of the utilities within the package. The utilities can be divided into two groups - those which perform *external* operations on files, and those which perform *internal* operations. In this context the word

external signifies that the operation is performed on the file as a whole - in other words, the file contents are irrelevant. The word *internal* means that the result of the operation depends upon the contents of the file.

The utilities which operate on the internal contents of files are:

- BUILD
- CXREF
- DIFF
- GREP
- SPLAT
- WC

The utilities which operate externally on files are:

- EXTRACT
- FILES
- LMK
- LPROF
- LSTAT
- TB
- TOUCH

A brief description of each utility is shown below:

BUILD	Builds a command file from a list of filenames.
CXREF	Generates a cross-reference listing of a set of C language source files.
DIFF	Compares files and reports differences.
EXTRACT	Extracts filenames from a directory.
FILES	Locates files by specified attributes.
GREP	Matches character patterns within a file.
LMK	Automates the management of files based upon dependencies and rules.
LPROF and LSTAT	Provides profiler and reporting capabilities for performance tuning.
SPLAT	Searches and replaces within one or more files.

TB	Provides a trace-back capability for complex problem debugging.
TOUCH	Updates the date and time stamps of individual files.
WC	Counts the number of words within a file.

1.2 Hardware Requirements

The Lattice Programmer Utilities require the following minimum hardware configuration for operation:

- 512K of main system memory
- AmigaDOS Version 1.2 or later
- 880K 3.5-inch disk drive
- An RGB(A) color display monitor

Note that a memory expansion and a hard disk will significantly improve the overall performance of your system and simplify operations using any of these utilities.

1.3 On-Line Assistance

Note that many of the utilities feature some form of built-in help. Entering the name of the utility will provide a screen listing of the various options available with the utility.

For example, the **GREP** utility will give this screen:

```
Too few arguments to grep

grep [>outfile] [flags] pattern files ...

Flags:
    -$    make case insignificant
    -c    print only a count of the match lines
    -f    print only names of files in which a match was found
    -n    shut off line numbering
    -p    filter input for printable characters
    -q    don't display file names or line numbers
    -s    show name of each file as it is being searched
```

```
-v    print only lines in which a match is not found
-V    print version of grep
```

Section 2

BUILD and EXTRACT

BUILD and **EXTRACT** are two utilities which are used for quickly constructing batch command files. **EXTRACT**, as the name suggests, extracts filenames from a specified directory. **BUILD** causes the insertion or interleaving of lines of text in a given file. When used in conjunction with each other they comprise a powerful tool for the automating of complex tasks. The resultant batch command file can then be executed from the AmigaDOS CLI prompt.

2.1 Using EXTRACT

The command line syntax for **EXTRACT** is:

```
extract [-b | -r] [-n] file1 file2 ...
```

where:

-b Refers to the **basename** and means only the root portion of the filename is output. Any filename extension (suffix) is ignored.

-n Refers to the **no sort** flag and prevents the list of filenames from being sorted. By default, the filenames are sorted in an alphabetical order.

-r Refers to the **root** flag and indicates that only the root portion of the filename is to be output. In other words, any prefix or suffix is removed from the filename before being output.

Note that all of these flags are optional and wildcard characters is permitted for the filenames. For example, the command:

```
extract #?.info
```

results in each filename with an **.info** extension being printed on your screen (one name per line) from the current directory. Specifying a device and directory is also supported. Thus, to redirect the output to a file named **beta** located in a directory named **omega** on **dh0:**, use:

```
extract >dh0:omega/beta #?.info
```

The file **beta** would then contain the names of each **.info** file within the current directory. Using the same example, redirection to the printer, parallel, or the serial ports takes the following form:

```
extract >prt: #?.info
extract >par: #?.info
extract >ser: #?.info
```

Note that the command line redirection of output is handled by AmigaDOS and not by **EXTRACT**. This is why the redirection character must appear as the second parameter on the command line.

2.2 Using BUILD

The command line syntax for **BUILD** is:

```
build >alpha beta
```

where **beta** is assumed to be a file consisting of a number of lines of text and **alpha** is the generated output file. Typically, each line will be the name of a file if you are using **EXTRACT** and **BUILD** to construct batch command files.

For the sake of simplicity, we call each individual line of text within the file **beta** a name.

BUILD reads from the standard input device (usually the keyboard) and will wait (without displaying a prompt) for input from the keyboard. Each line of input from the keyboard is considered as a model from which to construct lines which are then printed to the output file named **alpha**.

The completion of your input is indicated by pressing the [Ctrl][\] keys and the RETURN key on the last line.

A model is a sequence of text lines where the exclamation point **!** and forward slash **/** have special interpretations. Each line within a model in which an exclamation point **!** appears is repeated for every name in the name list, and the corresponding name replaces the **!** character.

Within any model line, the forward slash indicates that a newline character is to be generated at this point. This feature allows multiple output lines to be generated for each name. To use a literal forward slash, you must escape it with a second one, hence **//**.

Model lines are read one at a time from standard input, and the name list file is rewound to the beginning of the file. In other words, you restart at the top of the file after each sequence of output lines is generated from a given model line.

2.3 Examples

The following examples are specially designed to teach you the concepts of using **EXTRACT** and **BUILD**. Please read through these examples; they are meant to be both demonstrative and interactive. Create some dummy files using the names given and then execute the source code material. This is the best way of learning about these tools.

2.3.1 Example 1

This example using **BUILD**, **EXTRACT** and **DIFF** will take files with like names from two directories. Note that **DIFF** is discussed later in this manual - refer to the section dealing with **DIFF** for further details.

Consider where we have two directories named **version1** and **version2** containing the following files:

version1	version2
main.c	main.c
file1.c	file1.c
file2.c	file2.c
	file3.c

You want to verify that each file in the directory **version1** matches its corresponding file in directory **version2**.

The command line syntax would look like this:

```
extract >files.txt -b version1/#?.c
build >compare files.txt
diff version1//! version2//!
Ctrl\
execute compare
```

The first line shown below:

```
extract >files.txt -b version1/#?.c
```

extracts any filename from the directory **version1** with a **.c** extension and places the filename into the file named **files.txt**. The -b option signals that the file extension is to be included and that the directory is ignored. The contents of the file **files.txt** would look like this:

```
main.c
file1.c
file2.c
```

In the second line shown below:

```
build >compare files.txt
```

the file **files.txt** is to be operated on by **BUILD** together with any keyboard input, and the result is to be placed in a file named **compare**. Executing this line will result in no prompt being displayed and you should type in the third line:

```
diff version1//! version2//!
```

The third line shown above, introduces the **DIFF** utility as well as the forward slash and exclamation mark.

When you have entered this line, you would press the RETURN key. Note that the screen remains without a prompt and to complete the task, you must press the key sequence shown on the fourth line of Ctrl \. You must then press the RETURN key after this line and since the system prompt is now visible, this indicates that the end-of-file condition was reached on line 4.

The basic command line syntax for **DIFF** is:

```
diff new-file old-file
```

In explanation of line 3, you will recall that the forward slash indicates that a newline character is to be generated. However, in this example we do not want a newline character since the forward slash serves as an AmigaDOS directory separator. Therefore, the forward slash needs to be a literal character and we have escaped this by adding another forward slash. The exclamation mark means repeat the name of the file.

The contents of the file named **compare** would look like this:

```
diff version1/main.c version2/main.c
diff version1/file1.c version2/file1.c
diff version1/file2.c version2/file2.c
```

Our final line is a standard execution instruction of the file named **compare** and the results are displayed to the screen.

```
execute compare
```

You should be able to try this example by creating the directories named **version1** and **version2** and placing the appropriately named files in them.

2.3.2 Example 2

In this example we are going to construct a batch command file for manipulating some files for a simulated compilation using the **Lattice AmigaDOS C Compiler**. The first step is to ensure that you are at the CLI prompt and your current directory is the root of the RAM disk. We will assume that the following files are in your current directory:

```
main.c
graphics.c
```

Our plan is for the files listed above to be compiled, the object files to be copied to drive **df0:** and then deleted from the current directory.

Enter the following from the keyboard:

```
extract >alpha -r main.c graphics.c
```

This creates the file named **alpha**. Note the use of the **-r** option to strip off the file suffix. The file **alpha** contains:

```
main
graphics
```

We now need to invoke **BUILD**. Enter this line from the keyboard:

```
build >beta alpha
```

Finally, the following would be entered from the keyboard:

```
lc1  -i//lc// ! /lc2 !
copy !.o df0:!.o
delete !.o
Ctrl \
```

Note how input is terminated with the Ctrl\ sequence. If you examine the file **beta**, you will find it contains the following commands:

```
lc1 -i/lc/ graphics
lc2 graphics
lc1 -i/lc/ main
lc2 main
copy graphics.o df0:graphics.o
copy main.o df0:main.o
delete graphics.o
delete main.o
```

Our current directory now contains the following files:

```
alpha
beta
main.c
graphics.c
```

Although the examples above are relatively simple, they serve to illustrate the basic features of **EXTRACT** and **BUILD**. Do not be afraid to experiment with more complex applications.

Section 3

CXREF

CXREF will generate a cross-reference listing from a set of C language source files. Note that **CXREF** is distinct from the **-gx** option in the compiler in that **-gx** creates a cross-reference on only one file, where **CXREF** will create a cross-reference on many files. The contents of the listing for **CXREF** are:

- Tabular Information about:
  - Symbols
  - Identifiers
- Line Numbers

3.1 Using CXREF

Output from this utility is sent to the standard output device which means that you can redirect the output to other devices, rather than the screen. The basic command line syntax for **CXREF** is:

```
cxref >destination source
```

The following command line provides an example of this:

```
cxref >beta alpha
```

This will operate on the file **alpha** to produce a file named **beta** within the current directory.

```
cxref >prt: alpha
```

In the above example, output from **CXREF** is sent directly to the printer port.

CXREF also supports the **path** directive for **INCLUDE** files. The command line syntax used for this is similar to that used for the **Lattice AmigaDOS C Compiler**. A major difference is that in the **CXREF** command, the options come after the file name.

For example:

```
cxref alpha.c -i/path/path
```

Note that more than one **-i** is permitted in a command line. Thus, the following is permissible:

```
cxref alpha.c -idf0: -idir1 -idir2 -i...
```

AmigaDOS wildcard nomenclature can be used with **CXREF** with a maximum of five filenames on each command line. Due to the range of wildcard options offered, it is possible to use this feature to cross-reference an entire project involving many files.

An example of a command line using AmigaDOS wildcards is:

```
CXREF #?.c #?.h a#?.#? [option flags]
```

The source listing prefixes each line with a line number and recognizes three special comment constructions. Each construction has the following effects within the output listing:

/** This will provide a forward slash followed by a row of asterisks extending the full width of the page.

**/ This is the reverse of the above and provides a line of asterisks extending the full width of the page. The final character is a forward slash.

/**/ This will result in a form feed character to be sent to the output file. In effect, a new page is started.

The cross-reference section of the output produces individual tables for the following categories of symbols:

- Pre-processor Defined Identifiers.
- Functions.
- Labels.
- Structure Identifiers.
- Identifiers.

A typical table takes the form:

```
==>> Functions:
main
    cat.c   :      17
    ftoc.c  :      8
printf
    ftoc.c  :      18
```

The first line gives the table type, in this case it is the functions table. The following line names the function, in this case the function is **main**. Lines 3 and 4 provide the name of the file where this function occurs and the line number on which the function appears. The same sequence is followed for the next function which is **printf**.

CXREF also provides a large selection of options to control various aspects of the utility.

These options are:

-ipath This option causes **#include** files to be processed for both listings and cross-reference information. When this option is

selected, filenames will appear in both parts of the output. This takes the form:

aaa.rrr

where **aaa** represents the absolute line number and **rrr** refers to the relative line number within the file. In the case of the former, the absolute line number means the total number of lines preceding this line, including those in the **#include** files. Both **path** and **INCLUDE:** are searched when **path** is unspecified with this option.

-lnn This option defines the page length, which has a minimum value of 10 lines per page. The default value is 66 lines per page using standard sized printer characters. The **nn** represents the lines per page value.

-n This option removes page headings from the listing. By default, **CXREF** will generate a form feed and place a heading on each page.

-o This option omits identifiers. This can be a useful memory conservation device, particularly with large programming projects.

-p This option generates a program listing with **no** cross-reference report.

-r This option causes any C language reserved words, such as **if** or **for**, to be included in the cross-reference. When this option is selected two additional categories of symbols are listed:

- Pre-processor Commands.
- Reserved Keywords.

-wnn This option controls the page width. The default value is 80 characters per line. The minimum page width value is 60 characters per line; the maximum value is 132 characters. The "nn" represents the number of characters on a line.

-x This option will generate a cross-reference report. No program listing will be produced.

3.2 Examples

This command line shown below will generate a listing and cross-reference report for the file **alpha**. No form feeds or page heading will be output. Any **#include** files will be processed and C language reserved words will be listed in the cross-reference report.

```
cxref alpha -i -r -n
cxref beta -l 60 -x
```

The above command line generates a cross-reference report for the file **beta** without a program listing. Form feeds and page headings will be output, with a page length of 60 lines.

A typical program listing would look like this:

```
CXREF (v1.0) - `C' Cross Reference for: cat.c      Page 1

1-cat.c   -   1    /* This program is from K&R book */
2-cat.c   -   2   #include <stdio.h>
3-cat.c   -   3
4-cat.c   -   4   main(argc, argv)  /* concatenate files */
5-cat.c   -   5   int argc;
6-cat.c   -   6   char *argv[];
7-cat.c   -   7   {
8-cat.c   -   8      FILE *fp, *fopen();
9-cat.c   -   9
10-cat.c  -  10     if (argc == 1) /* no args */
11-cat.c  -  11          filecopy(stdin);
12-cat.c  -  12     else
13-cat.c  -  13       while (--argc > 0)
14-cat.c  -  14        if ((fp=fopen(*++argv,"r"))==NULL){
15-cat.c  -  15           fprintf(stderr,
16-cat.c  -  16            "cat: can't open %s\n", *argv);
17-cat.c  -  17           exit(1);
18-cat.c  -  18      } else {
19-cat.c  -  19           filecopy(fp);
20-cat.c  -  20           fclose(fp);
21-cat.c  -  21        }  +
22-cat.c  -  22     exit(0);
23-cat.c  -  23   }
24-cat.c  -  24
```

```
25-cat.c  -  25  filecopy(fp)  /* copy file to stdout */
26-cat.c  -  26  FILE *fp;
27-cat.c  -  27  {
28-cat.c  -  28      int c;
29-cat.c  -  29
30-cat.c  -  30      while ((c = getc(fp)) != EOF)
31-cat.c  -  31          putc(c, stdout);
32-cat.c  -  32  }

==>> Functions:

exit            :    17    22
fclose          :    20
filecopy        :    11    19    25
fopen           :     8    14
fprintf         :    15
getc            :    30
main            :     4
putc            :    31

==>> Identifiers:

EOF             :    30
FILE            :     8    26
NULL            :    14
argc            :     4     5    10    13
argv            :     4     6    14    16
c               :    28    30    31
fp              :     8    14    19    20    25    26
stderr          :    15
stdin           :    11
stdout          :    31

*******************************************************
CXREF (v1.0) - `C' Cross Reference for: ftoc.c    Page 1

1-ftoc.c  -   1   /* print Fahrenheit-Celsius table
2-ftoc.c  -   2        for f = 0, 20, ..., 300
3-ftoc.c  -   3        (This program is from K&R book)
4-ftoc.c  -   4         */
5-ftoc.c  -   5  int lower, upper, step;
6-ftoc.c  -   6  float fahr, celsius;
7-ftoc.c  -   7
8-ftoc.c  -   8  main()
```

```
9-ftoc.c  -   9  {
10-ftoc.c -  10
11-ftoc.c -  11   lower = 0;  /* lower limit of temp */
12-ftoc.c -  12   upper = 300;       /* upper limit */
13-ftoc.c -  13   step = 20;         /* step size */
14-ftoc.c -  14
15-ftoc.c -  15   fahr = lower;
16-ftoc.c -  16   while (fahr <= upper) {
17-ftoc.c -  17    celsius = (5.0/9.0) * (fahr-32.0);
18-ftoc.c -  18    printf("%4.0f %6.1f\n",fahr,celsius);
19-ftoc.c -  19    fahr = fahr + step;
20-ftoc.c -  20   }
21-ftoc.c -  21  }

==>> Functions:

main              :      8
printf            :     18

==>> Identifiers:

celsius           :      6     17     18
fahr              :      6     15     16     17     18     19
lower             :      5     11     15
step              :      5     13     19
upper             :      5     12     16
```

Section 4

DIFF

DIFF is a utility which will determine the difference in contents between two files and then display these variations in an informative way. Note that a discussion of the theory of file matching and how this impacts **DIFF** appears in Appendix B.

For example, consider two files named **alpha** and **beta** where we want to compare the contents of the former with the latter file. The contents of file **alpha** are:

```
Mary had a little lamb.
Its fleece was white as snow.
```

File **beta** contains the following:

```
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went
the lamb was sure to go.
```

The output from **DIFF** takes this form:

```
*** APPEND AFTER 2 IN alpha ***
>   And everywhere that Mary went
>   the lamb was sure to go.
```

DIFF shows the differences and in this case the output from **DIFF** informs us that after the second line, the displayed text should be appended. Refer to the subsection titled **Output Format** for further details.

4.1 Using DIFF

The command line syntax for **DIFF** is:

```
diff [options] newfile oldfile
```

DIFF normally responds by directing output to the screen. Output may be redirected to a file or device in the usual AmigaDOS fashion or by using the **-o** option described below. The basic command:

```
diff
```

results in a usage message being printed informing you of the options available and the version of **DIFF** you are using.

4.2 Options

Several options are available for use with **DIFF**. These are:

-bnn	This option will set the size of the I/O buffer in bytes, to the value specified in **nn**; the default is 4K. The maximum size is limited by the amount of available system memory.
-c	This option will display only lines common to the files under comparison.
-Fnn	This option will set the position of the first column in the file to be compared. This means if you specify the value of **nn** as being 10, the comparison will ignore any characters to the <u>left</u> of the tenth character.

-Lnn This option will set the position of the first column in the file to be compared. This means if you specify the value of **nn** as being 10, the comparison will ignore any characters to the **right** of the tenth character.

-lnn This option will determine the size of the table which **DIFF** uses for the lines in each file to **nn**. By default, **DIFF** assumes that the files being compared will have a maximum of 2000 lines in each file and creates two tables (one for each file) of size 2000. In situations where the files you are comparing are larger than 2000 lines, you may use this option to increase the size of the two tables. If you seem to be running out of memory in comparing your files and the files are less than 2000 lines in length, you may use this option to decrease the size of the tables and leave more memory available for **DIFF** to use. Alternatively, check to see if any other tasks are running concurrently and shut down any unwanted tasks.

-o file This option will direct the output from **DIFF** into the specified file. Output device options are available by prefixing the filename with the device e.g. **df0:** or **prt:**, etc.

-p This option will filter any printable characters. This option is identical to the **-p** option of **GREP** and removes non-printable characters from the input stream. This is useful for cleaning out an AmigaDOS **binary** file.

-q This option will not display any messages when there are no differences between the files under comparison.

-w This option will make **DIFF** ignore differences which relate only to whitespace (spaces or tabs). When two lines are compared using this option, trailing blanks are removed and sequences of whitespace are compressed to a single space.

4.3 Output Format

The output format of **DIFF** appears as follows. After giving the command:

```
diff newfile oldfile
```

DIFF initially responds by displaying:

```
TO TRANSFORM oldfile INTO newfile ...
```

followed by a sequence of three different types of **change blocks**.

The three types of change blocks are:

- Delete
- Append
- Change

Delete A **delete** block of the form:

```
*** DELETE [i,j] FROM oldfile ***
<aaa
<bbb
<ccc
```

where **aaa, bbb, ccc,** etc. will be the text of the lines to be deleted from **oldfile**, and **i** and **j** are the first and last line numbers in **oldfile** of the block to be deleted. The less than symbol (<) preceding the line text indicates that a deletion is to take place.

Append An **append** block of the form:

```
*** APPEND AFTER i IN oldfile ***
>aaa
>bbb
>ccc
```

where the greater than symbol (>) indicates that an append operation is to take place, and **i** is the line in **oldfile** after which the lines represented by **aaa, bbb, ccc,** etc. are to be appended.

Change A **change** block of the form:

```
*** CHANGE [i,j] IN old TO [m,n] IN new ***
<aaa
<bbb
<ccc
---------------
>xxx
>yyy
>zzz
```

where the block of lines (numbers **i** to **j** in **oldfile**) represented by **aaa, bbb, ccc**, etc. are to be changed to the block of lines (numbers **m** to **n** in **newfile**) represented by **xxx, yyy, zzz**, etc. This replacement is equivalent to a deletion of the first block and an appending of the second.

4.4 Error Messages

The following error messages may be generated by **DIFF**.

Can't open ...

DIFF is unable to open the named file. This could be because the file does not exist or because it is protected.

Diff I/O Error

Although an I/O problem is the usual cause of this message, you should be aware that this is a general-purpose error message from **DIFF**.

Improper -l specification: ...

The specification given to the **-l** option cannot be interpreted as a number by **DIFF**.

Internal Error: ...

If you see a message of this sort please contact the Lattice Technical Support Department and be prepared to provide the following information:

- The version of **DIFF** you are using.
- The exact wording of the command line you used.
- The exact wording of the message you received.
- The size of the files you were trying to compare.

Line table overflow

The line table is too small to hold the number of lines in one of the files being compared. Try increasing the line table size with the **-l** option.

Not enough memory for ... lines per file

You have used the **-l** option to increase the line table size, but you do not have enough memory to increase it by the amount specified.

Out of memory

Try decreasing the line table size with the -l option if this is possible. Do not use the **-w** option.

Too many file names: ...

You have made an error on the command line in such a way that **DIFF** assumes you are attempting to operate on more than two files.

Unrecognized option

You have attempted to invoke an option which **DIFF** does not recognize.

Section 5

FILES

The **FILES** utility is a powerful tool which allows you to search for, copy or erase files and directories. In some respects it is similar to the UNIX **find** utility, but in addition to the file searching ability, **FILES** has the functionality of the UNIX **cpio** and **rm** commands. This is reflected in the ability of **FILES** to copy, move, or remove entire directory structures.

The **FILES** utility may be used to search one or more directories for objects - either files or directories which are:

- Matching a specified **name** or **pattern**.
- A specific type - a directory or file.
- Modified within a specified number of days.
- **Younger** than a specified file or directory.
- **Older** than a specified file or directory.
- **Smaller** than a specified size.
- **Larger** than a specified size.

This utility is particularly useful if you know you have a file located somewhere on your disk but have forgotten exactly where it is - a common problem when using high-capacity hard disks. Even if you can only recall a part of the file's name, the **FILES** utility can locate the file.

In addition, **FILES** will allow you to copy entire directory structures from one subdirectory to another or from one drive to another. The utility also supports a **recursive erase** feature which allows you to remove an entire directory structure and its contents with a single command.

> NOTE: When a file or directory has been **assigned** by AmigaDOS - such as your **startup-sequence** file, or by a direct command through the CLI, the file or directory **cannot** be deleted by the **FILES** utility. To delete the required file or directory, you must re-assign the AmigaDOS assignment to another file or directory.

5.1 Using the Files Command

The general form of the **FILES** command is:

```
files [options] dir1 dir2 ......
```

where:

dir1, dir2 Represent directory names.

options Represents a sequence of options.

FILES not only searches the specified directories but also recursively searches all subdirectories of the specified directories, all subdirectories of those subdirectories and so on.

For example, assume your current directory is *Workbench* in **df0:** drive. To display every filename and subdirectory in the **devs** (devices) directory, the command line syntax used is:

```
files devs
```

This results in a listing such as:

```
Files Version 1.00: Copyright 1985, 1988 by Lattice, Inc.
devs/clipboards
devs/system-configuration
devs/clipboard.device
devs/narrator.device
devs/parallel.device
devs/printer.device
devs/serial.device
devs/keymaps
devs/keymaps/gb
devs/keymaps/usa0
devs/printers
devs/printers/laser
devs/printers/generic
```

Similarly, the command:

```
files /fonts .
```

will generate a listing of all files in both the directory named **fonts** and the current directory. Note the use of the period (.) for this variation of defaulting to the current directory.

5.2 Options

The **FILES** utility has a large number of features which make the manipulation of single or multiple files very simple. For example, you may not need a listing of every file in a directory, only a listing of certain files in that directory with attributes beyond the capabilities of the wildcard protocols. The **FILES** utility provides a number of options to accomplish this goal. Except where specifically stated, these options may be used in conjunction with one another.

-b This option is identical to the **-b** option in **Extract** and produces the base names of files.

-copy destn This option allows you to copy files or directories from one location to another. For example, the command:

```
files -copy ram: /fonts
```

will copy to **ram:** all files in the directory **fonts** (and all files in subdirectories of **fonts** subdirectories, etc.). The command will make any new directories or subdirectories necessary at the destination and copy all files. Note the space after the colon of the destination device and before the forward slash. Similarly, the command:

```
files -name #?.pic -copy /color /paint
```

will copy all files with **.pic** extensions which are anywhere under the directory **paint** to the directory **color** making subdirectories as necessary.

In the above example, the directory **colors** must already exist or **FILES** will display an error message that it cannot create the new (copied) versions of the files.

With this option, **FILES** provides much of the functionality of the UNIX **tar** or **cpio** commands. Used with the **-erase** or **-rerase** options, the **-copy** option may be used to **move** directory trees since the copy is done prior to the erasure.

-days nn

This option will search the current directory for files with a date stamp specified in **nn**. Any positive (decimal) number may be used as the **nn** parameter to the **-days** option. For example, you want to search the current directory and the directory **/x** for files which have been created or modified within the last five days, use the command:

```
files -days 5 . /x
```

-erase

This option causes **FILES** to erase the files which are selected. The **-erase** option will not descend into subdirectories, but will erase files only at the highest level of the specified directory. For example, the

command:

```
files -erase -name #?.src /lattice
```

will remove all **.src** files from the directory **/lattice.**

-m This option will instruct **FILES** to use wildcards. For example:

```
files -m -name *.pas
```

would search for all files with the extension **.pas** in the current directory.

-n This option forces **FILES** to refrain from recursively descending into subdirectories to do its work whether it be finding, copying or erasing.

-name file/dir This option will search for files which match a specified name. For example, the command:

```
files -name alpha.h
```

will look for a file whose name is **alpha.h** in the current directory. Note that the case of filenames is ignored by AmigaDOS. Similarly, **FILES** is compatible with this usage, and will detect files whether they appear as **ALPHA.H, alpha.h, Alpha.h,** etc.

Similarly, the command:

```
files -name #?.c tools/tmu
```

will search for all files with **.c** extensions in the subdirectory **tools/tmu** of the current directory. The command:

```
files -name /tmu /tools
```

will list all files found in any subdirectory (sub-subdirectory, etc.) called **tmu** of the directory **/tools**.

The filename specified with the **-name** option may contain the standard AmigaDOS wildcard characters and these will be interpreted in their usual sense.

-newer This option will locate all files or directories which have a time stamp later than the specified time stamp. For example, to locate any files in the subdirectory named **bugs** which are newer than the file **/src/test.c**, the command used would be:

```
files -newer /src/test.c bugs
```

-older This option will locate all files or directories which have a time stamp earlier than the specified time stamp. Both this and the previous option use the file or directory creation date as the operating parameter.

-pat This option will locate files using the pattern matching facilities of **GREP**, rather than AmigaDOS and DOS wildcards. These patterns are more comprehensive than those allowed by AmigaDOS and DOS. Thus the command:

```
files -pat text[0-9] /docs
```

will search the directory **/docs** for any file whose name contains the word **text** followed by a decimal digit. In using the **-pat** option you must remember that a number of characters such as "." "+" "*" are interpreted in a special sense. Refer to the section detailing **GREP** for more details on these patterns.

In common with the **-name** option, this option translates the specified pattern to uppercase prior to beginning its search.

Note that the **-pat** and **-name** options are incompatible with each other.

-r This option is identical to the **-b** and **-r** options in **EXTRACT**, producing base and root names of files.

-rerase This option will cause **FILES** to erase the selected files. The **-rerase** option will recursively descend into subdirectories and remove all selected files.

For example, the command:

```
files -rerase -name #?.c /lc
```

will remove all **.c** files anywhere under **/lc** or its subdirectories, sub-subdirectories, etc. Finally, the command:

```
files -rerase :
```

will remove all files on your current disk drive. This is equivalent to the UNIX **rm -rf *** command.

-size nn This option allows you to search for files specified by the number of bytes **nn**. For example, to search for files which are at least 20K bytes in size in the current directory, use the command:

```
files -size 20000
```

-type arg This option allows you to restrict your search to files of a given type. There are two arguments to this option. These are:

d directories
f normal files

This option allows you to distinguish between files and directories.

The command:

```
files -type f . /docs
```

will list only the files in the current directory and the **docs** directory. The command:

```
files -type d . /docs
```

will list only the (sub)directories in these directories and recursively in their subdirectories. The **-type f** and **-type d** options are incompatible with one another.

-v This option will cause **FILES** to list the files and directories it is removing to the screen.

Many of the options may be used in conjunction with one another. For example, you may copy and erase files and thus, move them, find files newer than one file and older than another, and so on. If your selection of options cannot be executed, **FILES** will issue an error message to this effect.

Section 6

GREP

GREP (**G**lobal **R**egular **E**xpression Search and **P**rint) is a utility which searches a set of files for a specified character pattern and prints each line containing the expression matching the pattern. **GREP** is a powerful tool of great value to software developers, technical authors, as well as many personal computer users because it is simple to use. It can save a great deal of time and help minimize the frustration which often results from trying to find words or sequences of words which may occur in a number of files.

The companion program **SPLAT** makes considerable use of **GREP**'s pattern matching abilities. These abilities allow you to develop editing of script files and make complex changes such as addition, deletion and substitution to a set of files in an automated way.

We begin our discussion of **GREP** with a simple example and proceed to explore the intricacies of **GREP**.

6.1 Using GREP

The general form of a **GREP** command is:

```
grep [flags] pattern file1 file2 ...
```

where:

pattern The pattern you wish to search for.

file1 file2 The files you wish to search.

The **flags** parameter is discussed later in this section. Note that the enclosing square brackets ([ ]) mean that the flags parameters are optional.

The patterns **GREP** can search for are quite general and are sometimes referred to as **regular expressions**. A full and precise description of patterns is given later in this section.

6.2 A Simple Example

We will assume that you are a technical author who has been editing a file, **document.txt** and the file has become quite long. It occurs to you, in certain contexts where you should have used the phrase **user manual**, you actually used the phrase **user's guide**. You would like to determine how many occurrences of this error exist and where they occur in your file in order to correct the error.

The correct command to use is:

```
grep "user's guide" document.txt
```

The entire phrase must be enclosed in double quotation marks (" ") to inform **GREP** of the correct phrase to search for. The double quotation marks are necessary in this command line since we are searching for a phrase separated by space characters. In situations where we would be looking for a single word, then the double quotation marks are unnecessary. When the quotation marks are not used, for example in the following command line:

```
grep user's guide document.txt
```

GREP interprets this as a request to search for the word **user's** in two files

named **manual** and **document.txt**. You need to be quite specific in describing the search pattern to **GREP**. For example, using the following command:

```
grep manual document.txt
```

GREP will respond by printing all of the lines in the file which have the word **manual** in them. However, this is not quite the desired answer since there are likely to be other lines which contain the character sequence **manual** in addition to the phrase **user's manual**. For example, there could be phrases such as **this manual, the manual, you must insert the paper manually** and so on, within your file.

6.3 Special Characters

Before we describe the different kinds of patterns, you should appreciate that there are a number of characters which **GREP** interprets in a special way.

For example, if you were to use the command:

```
grep [ document.txt
```

you would receive an error message from **GREP**. This is because the left bracket ([) has a special meaning to **GREP**. This subsection describes the range of special characters, later subsections will describe how **GREP** interprets each of these special characters. This subsection will also show you how to inform **GREP** that you do not want the character interpreted with their special meaning.

Each of the following characters is regarded as a special character to **GREP**:

! $ ^ * - + [ ] . \

The last of these characters, the backslash (\) is called the escape character and is used to escape any special character (including itself) and cause **GREP** to ignore the special nature of the character.

Using the example from the previous subsection, if you wanted to search for the left bracket you would use the command:

```
grep \[ document.txt
```

or if you wanted to search for the phrase **'and so on ...'** you would use the following command:

```
grep "and so on \.\.\." document.txt
```

To search for the backslash (\) itself use the following:

```
grep \\ document.txt
```

If a **GREP** command does not respond as you expect it to, try escaping any special characters in the command with the backslash \ character. Alternatively, try reducing the number of the special characters being used in your search pattern.

6.4 Escape Sequences

In addition to these special characters, there are several sequences of characters (escape sequences) which **GREP** treats in a special way. Each of these begins with a backslash followed by a single letter and their meanings are as follows:

\n	newline character
\s	space character
\b	backspace character
\t	tab character
****	escape character
\xij	hexadecimal number whose digits are *i* and *j*

For example, to locate all lines containing tab characters you would use the command:

```
grep \t document.txt
```

The last escape sequence is provided for those circumstances in which you might wish to search for the occurrence of a character by using its hexadecimal number. Control characters are generally used as printer format instructions in files produced by word processors. For example, to search for occurrences of a Ctrl G character, the command line would be:

```
grep \x07 document.txt
```

Determining the value of the control character is simply a matter of using the position of the letter in the alphabet. For example, **C** is the third letter of the alphabet, thus the Ctrl C character is represented by 03. To search for all occurrences of the Ctrl C character, the command line would read:

```
grep \x03 document.txt
```

Similarly, **Z** is the twenty sixth letter of the alphabet, which equates to 1A in hexadecimal. Thus, to locate any Ctrl Z characters, the command line would be:

```
grep \x1a document.txt
```

6.5 Simple Patterns

The simplest patterns are the types we have just seen examples of - words or strings of letters and phrases possibly containing spaces or tab characters. To search for a single word it is sufficient to use just the word as the pattern. However, to search for a string of words which are separated by spaces you must enclose the entire string within **double** quotation marks as in the following command:

```
grep "this is the string to search for" document.txt
```

6.6 The Wildcard Character

The special character, the period (.) is regarded as a wildcard character by **GREP** in that it is taken to be matched by any single character. Thus the pattern:

```
.he
```

will be matched by any of the following:

ahe **bhe** **che** **she** **the**

Note that the dot will be matched by any single character, not just by any single alphabetic or numeric character. Thus it will be matched by a space, tab, newline or control character. The wildcard character is one of those you must carefully consider when you are constructing patterns.

For instance, if you wanted to see the lines in your file which contain the expression **123.45**. The command you must use is:

```
grep 123\.45 document.txt
```

where the period (.) is escaped.

The pattern:

```
123.45
```

would have matched any of the following expressions:

123a45   123 45   123!45   123(45   123.45

6.7 Character Classes

A *character class* is a pattern which is matched by any character in a given category or set of characters. **GREP** uses the left and right bracket symbols ([ ]) to mark the beginning and end of a character class pattern. For example, the expression:

```
[abc]
```

denotes a character class which is matched by any of the first three (lowercase) letters of the alphabet and the expression:

```
[Tt]
```

denotes a character class which is matched by either an uppercase **T** or a lower case **t**.

Character classes are useful in situations where you want to construct a certain type of disjunctive pattern. In other words, you want to search for groups of characters where the case of an individual character is unimportant. For example, you wish to view the lines in your document containing either word **the** or **The**.

Of the following commands:

```
grep the document.txt
grep The document.txt
grep [Tt]he document.txt
```

The first will only print out the lines containing the word **the**, the second will print out only lines containing **The**, and the third will print lines containing both **the** and **The**.

Similarly, the command:

```
grep [Tt][Hh][Ee] document.txt
```

will print any line in which any of the following occur:

THE THe The the thE tHE tHe ThE

Within a character class, certain symbols are interpreted in special ways. You may use a minus sign - to indicate a *range* of characters, as in:

```
[a-z]
```

which is matched by any lowercase letter in the English alphabet or:

```
[a-zA-Z0-9]
```

which is matched by any lowercase letter, any uppercase letter or any decimal numeral.

A simple, yet practical use of character classes arises when you want to view all the lines in your file where you refer to a specific year. The command:

```
grep [0-9][0-9][0-9][0-9] document.txt
```

would accomplish this.

Where you are concerned with years only in the 20th century then use:

```
grep 19[0-9][0-9] document.txt
```

If you are concerned with years in the 19th and 20th centuries then use the following command:

```
grep 1[89][0-9][0-9] document.txt
```

The exclamation point (!) is regarded as special when it occurs as the **first** character in a character class. In this situation it acts as a **negation operator**, thus the pattern:

```
[!a-z]
```

will be matched by any character which is not a lowercase letter of the English alphabet, and the pattern:

```
[!#]
```

will be matched by any character which is not a number sign. If an exclamation point (!) occurs in a character class elsewhere than in the first position, it does not have this special sense.

Character classes lend a great deal of power to **GREP** and allow you to search for some very complex patterns.

6.8 Closure Characters

The symbols, asterisk (*) and plus sign (+) are used to denote *closures*. The asterisk (*) may be considered as meaning **zero or more of**, while the plus sign (+) means **one or more of.**

Consider a file where you have referred to **Gail** and mentioned a year in which some event has occurred. You do not want to see every line in which the name **Gail** occurs, nor do you want to see every line in which a yearly date occurs. You want to see only those lines in which **both Gail** and an annual date occur. The command would then be:

```
grep "Gail.*[0-9][0-9][0-9][0-9]" document.txt
```

Within the pattern, the expression **.*** would be matched by any string containing **zero or more** occurrences of any character. The entire pattern will therefore match such expressions as:

> Call Gail Barbara Industries on 1-999-629-9310
> Gaillard de Marentonneau was an 1900s century botanist
> My eldest daughter Gail, was born in 1969

Where there is a choice to be made concerning how long the pattern is which matches a closure, **GREP** chooses to match the longest possible pattern. To

consider another simple example, suppose you want to view all the lines containing any of the following words:

the **The** **she** **She** **he**

If you use the pattern:

```
[tTsS]*he
```

it will be matched by the following examples:

tttthe **TsShe** **Sthe** **TTTSSStthe**

It would be useful to detect these obvious typographical errors in any event. However, notice that this pattern would be matched by <u>any</u> expression containing **he** because of the zero or more qualifier on the character class. A better pattern would be the following:

```
[!a-zA-Z][tTsS]*he[!a-zA-Z]
```

To eliminate those contexts where **he** is embedded in a longer word, or to ensure that the word is isolated by spaces use the following:

```
" [tTsS]*he "
```

The plus sign symbol (+) works precisely the same as the asterisk (*) except for its requirement that at least one occurrence of its character or character class be present.

Thus:

```
[tTsS]*he
```

will be matched by **he** but the following will not:

```
[tTsS]+he
```

6.9 Anchored Searches

Frequently you may wish to view only those lines in a file in which a given pattern occurs at the **beginning** of the line - in other words, beginning at the column 0 position. The carat (^) is a special character when it is the **first** character in a pattern. This signifies that the pattern will be matched only if it begins a line in the file. Thus, to find all lines within your file which begin with a space or a tab character, use the command:

```
grep "^[ \t]" document.txt
```

Note that the double quotation marks (" ") are necessary in this pattern since a space occurs within the pattern itself. Similarly, to find all lines which begin with numerals, for example, a section heading, use the following:

```
grep ^[0-9] document.txt
```

The special character, dollar sign ($) works much like the carat (^) except that it forces the match to occur only at the **end** of a line. Consider for example, where you wanted to search for all occurrences of the C language comment terminator */ at the end of a line, you would use the following:

```
grep \*/$ document.txt
```

Here, the backslash (\) is used to escape the asterisk (*) which otherwise would be interpreted in its special sense as a closure operator.

6.10 Advanced Examples

This subsection will provide some additional examples of using **GREP** to search files for patterns. It will concentrate on searching for files containing source code text, in particular, C language source code.

6.10.1 Example 1

You have a number of C language source files, each with filenames having a **.c** extension, and you need to search these files to find all calls of either of the functions **read** or **fread**. Use the following command:

```
grep "[f]*read[ ]*(" #?.c
```

This will cause every line containing either **read** or **fread**, followed by zero or more spaces, followed by a left parenthesis located in all the **.c** files to be printed out.

6.10.2 Example 2

Consider where you want to search your files for every function definition - however, this really is not possible unless you make some reasonable assumptions. It is common for most C programmers to define a function by first placing the name and parameter list of the function on a separate line. This is followed with the body of the function. Making this assumption, use the command:

```
grep "^[a-zA-Z0-9_]+[ ]*(" #?.c
```

The pattern above will match only an expression which begins a line. The expression must consist of one or more lower-case letters, upper-case letters or the underscore. This must be followed by zero or more spaces and then a left parenthesis. This pattern will

pick out lines such as the ones shown below:

```
getl(s)          /* get a line */
_read ()
my_read(fp)  { fread(fp,x);}
```

However, it will not select a line such as:

```
char *getl(p)        /* get a line */
```

which may be the beginning of a function definition.

6.10.3 Example 3

You are having problems compiling a C program because the compiler complains that there is a mismatch between left and right braces ({ }). The program is very long and you have been unable to determine where the problem lies. You decide to use **GREP** to assist by printing each line which contains either a left or right brace. In this way the error may be easier to locate. The command to use is:

```
grep [{}] #?.c
```

6.10.4 Example 4

Consider where you want to find all occurrences of strings in your files, for example, sequences of characters enclosed in double quotation marks (" "). The appropriate command is:

```
grep \".*\" #?.
```

6.11 Using the GREP Functions in a Program

The **GREP** utility makes use of several functions which construct an internal representation of a pattern from a string of characters and then checks to see if this pattern is matched in a given string. You are supplied with files on the distribution disk which allow you to make use of these functions in C programs. The functions are included in the library file, **grep.lib** (assuming that you are using the **Lattice AmigaDOS C Compiler**). This allows you to build the same sophisticated pattern matching capability into your own programs to the same levels as **GREP**. This subsection reviews the procedures to accomplish this goal.

The files in the **GREP** library are:

```
re_gen.o
re_match.o
re_smatc.o
```

These files contain the functions:

```
re_gen()
re_match()
are_match()
re_smatch()
are_smatch()
```

The usage of these functions becomes clear with a simple example. Consider a global array of strings consisting of pointers to characters. You require a function which will search this array for strings containing a match for a given pattern. In addition, assume that the pattern you wish to search for may be described as:

> **any sequence of characters, including the empty sequence enclosed in parentheses.**

Finally, assume that the size of the array is defined by the constant named **ARYMAX**. Your code should look something like this:

```
extern char *ary_str[ ];
/*  Assumed to be defined elsewhere */
/*  Define a function findpars() which will search ary_str
* for members containing a string of characters enclosed
* in parentheses. For each such member of ary_str,
* print its array index. */
findpars()
{
    int i;

    for ( i = 0; i < ARYMAX; i++ )
      if ( re_smatch("(.*)",ary_str[i]) >= 0 )

         printf("Found match at index %d\n",i);
}
```

This code will work, albeit inefficiently. Every time **re_smatch()** is called, it calls **re_gen()** to generate an internal representation of a pattern. In the example the pattern never changes, so why regenerate it each time around the loop?

A better definition of **findpars()** would be:

```
#include "pat.h"  /* Need to include this header */
                  /* to have* `PATTERN' defined */
findpars()
{
 PATTERN re_gen(); /* must declare re_gen() as */
                    /*   returning a PATTERN */
 PATTERN p; /* to point to the pattern generated */
 int i;

    if ( (p = re_gen("(.*)")) == NULL ) {
         printf("Error in generating pattern\n");
         return;
    }

    /* pattern was generated okay */

    for ( i = 0; i < ARYMAX; i++ )
      if ( re_match(ary_str[i],p) >= 0 )
        printf("Found a match at index %d\n",i);
    }
```

This example should suffice to demonstrate how the supplied functions may be used in programs where the pattern-matching power of **GREP** is required. Naturally, you must be sure to link in the appropriate object modules when you link your program together.

If you are using the **Lattice AmigaDOS C Compiler**, then the command line to the linker should appear:

```
LINK:blink FROM LIB:c.o+myprog.o TO myprog
     LIBRARY grep.lib+LIB:lc.lib+LIB:amiga.lib
```

assuming that **myprog.o** is the object module you have compiled which in-

vokes calls to functions in the **GREP** library. The command line above assumes that you are using one of the **startup-sequence** files provided with the **Lattice AmigaDOS C Compiler.** These files make logical name assignments to directories.

Note that the command line shown previously is shown as two lines due to its length - in reality, it would be entered as one complete line.

6.12 Error Messages

Bad character class

GREP is unable to interpret a character class in the pattern you have asked it to find. Check to see if you have used any special symbols which you did not intend to. This message can also be generated if in specifying a character class and using the '-' character to indicate a range of characters, the first character in the range does not alphabetically precede the second (e.g. **[d-a]**).

Bad pattern

GREP is unable to interpret the pattern you are asking it to find. Most likely this is a result of the occurrence of a special character in the pattern which does not make sense to **GREP** in this context. Check to see if you have forgotten to escape a special character.

Can't find file(s) ...

GREP is unable to find one or more of the files you have asked it to search. Most likely, they are not there or you have mis-typed the file names.

Can't open ...

GREP is unable to open the named file. This could be because the file is not there at all or that it is protected.

Closing] not found

GREP believes that you have asked it to search for a pattern that contains a character class but that you have omitted the right bracket (]) which terminates the character class. If this is not so, perhaps you wanted the left bracket ([) to form part of the pattern but forgot to escape it to remove its special sense.

Empty character class

A character class you have used in your pattern has no members, e.g. [ ].

Invalid option

You have used a command line option which **GREP** does not recognize. This message can also be generated if your pattern begins with a minus sign (-) but you have not escaped it nor enclosed the pattern in double quote marks (" ").

Improper hex specification

You have used the \x escape sequence but have not followed it with two recognizable hex digits.

Incompatible combination of options

The options with which you have invoked **GREP** are asking it to do something contradictory.

No beginning double quote in pattern

GREP has detected a double quote (") in a pattern which was not started with a double quote. Perhaps you meant to escape it.

No file arguments provided

GREP believes that you have invoked it with a pattern but no filenames. This can also happen if you have invoked it with a filename but no pattern.

No pattern or file arguments given

GREP cannot find either a pattern or filename on its command line.

Out of space

There are several out of space messages you may see. They mean that **GREP** has run out of space in constructing its pattern representation. Please inform our Technical Support Department if you see this message.

Pattern ill-formed: no terminating double quote

You started the pattern with a double quote (") but failed to terminate it with one.

Too few arguments to GREP

GREP demands at least two arguments on its command line, a pattern and at least one filename.

In addition to the above error messages, there are some internal error messages which you should never see. If you get any error message differing from the above, please inform our Technical Support Department, or send us a disk and a letter. You should be prepared to provide the following information:

- The version number of **GREP** you are using.
- The exact wording of the command line you used.
- The exact wording of the resulting error message.

Section 7

LMK

This section assumes you are familiar with the operation of the **Lattice AmigaDOS C Compiler**. In addition, those readers who are familiar with the concepts behind the UNIX **make** utility may wish to bypass the first two subsections of this part of the manual.

LMK is a tool which is used for maintaining projects composed of many files. A file can be programming source code, a data file for a graphics or audio, or perhaps a spreadsheet file. For example, a word-processing document of any size beyond a few pages is usually made up of sections, simply because it is easier to work with. Breaking the document file into modules means that loading/saving cycles are shorter, moving around the document is made easier and text manipulations, such as spelling checks and cut-and-paste operations are faster.

To summarize the above, breaking down a large file into smaller files has a number of advantages. However, there is a cost overhead to this - how do you keep <u>control</u> over these files? In other words, how do you remember which files to include and in what order, how do you keep track of files which have been amended? The Lattice utility named **LMK** will enable you to do this.

7.1 What is LMK?

The basic function of **LMK** is to generate a product file if any of the source files have changed since the last version of the product file was generated.

In essence, **LMK** captures and deploys the information about how to turn a collection of raw materials - namely the source code files into a finished product. The information on how this is to be done is contained within the description file or **lmkfile**.

The C programming language encourages the use of multiple compilation units. Without an **LMK** utility, you would have to be able to recall the names of the individual files which needed to be recompiled or relinked whenever a change was made to one of the files. A **lmkfile** specifies which pieces of a product are to be recompiled or relinked when a few of the pieces are modified. Thus, **LMK** automates many of the tedious details associated with generating a new version of a project.

7.2 Time Stamps

LMK operates on the basis of time stamps associated with each file under consideration. In other words, the time and date of creation of a file determine the operation of **LMK**. A file is considered to be current if it has a time stamp later than that of any of its **dependent** files.

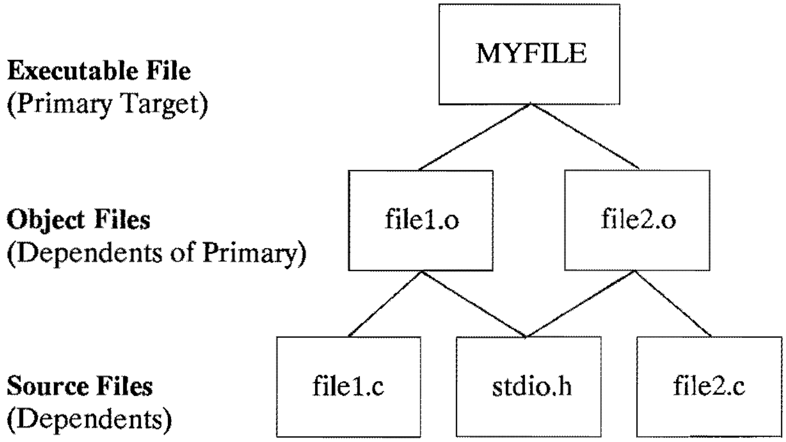
The syntax of the date command under AmigaDOS is as follows:

```
date 12-Nov-87 16:34:57
```

Note the semantics of the time section - this format is known as **Military Time** in the USA, or the **24-Hour Clock** in Europe and must be adhered to. Be sure to enter the month as a three character field.

> NOTE: It is extremely important to the successful operation of **LMK** that a consistently accurate current date and time are known to the system software.

7.3 Targets and Dependents



The diagram above outlines the concept of file dependencies and illustrates the interconnection between files. The **primary target** file serves as the end-result for the programming process. The object files are the intermediate step in the process - in this example the **primary target** file shows two dependent object files. Finally, we have the source code files - each object file has three dependent source code files. This means that our **primary target** file has a total of seven file dependencies.

A file created for use with **LMK** describes the relationships of all file modules associated with a specific project. In simple terms, **LMK** operates on the basis of time stamps associated with the dependent files in determining the required action(s) to be taken to rebuild the target files. **LMK** will perform these actions to update the target files.

LMK is useful in situations where more than one programmer will be working on a project. When a project is shared between programmers, the **LMK** file will contain all the necessary information describing the dependencies of the various source files. This permits more effective management of a programming team, even when a programmer is not on site or has left the project.

Each of the **dependents** of the target file under consideration is known as a **sub-target**. Hence we see that there are generations of files, each generation having certain descendant and ancestor files. The commands given to **make** the **target** and each of the **sub-targets** are known as **actions**.

7.4 Actions

So far, we have discussed the concepts of **targets** and **dependencies** in terms of **LMK**, we now examine in greater detail the final element, namely that of **actions**.

We discussed earlier that working in the C programming environment usually involves using several sets of files. The act of compilation also entails an additional operation called **linking**. In basic terms, this means joining or amalgamating your **object** code files with the compiler library files to form **executable** files. **Object** files are intermediate files produced by the **Lattice AmigaDOS C Compiler**.

The command line syntax for linking and compiling is very precise and any errors will cause unwanted results. A typical command line for linking looks like this:

```
blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.lib
```

It should be obvious that considerable scope exists for typing errors, particularly if such lines have to be entered each time you wish to link and compile files. To avoid this, the command lines could be encapsulated into a **lmkfile**.

7.4.1 A Simple Example

A simple **lmkfile** would look like this:

```
myfile: file1.o
  blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.lib

file1.o: df1:source/file1.c headers.h
   lc -b0 -r0 -odf1: df1:source/file1.c
```

This **lmkfile** demonstrates a linking and compilation sequence and is made up of four lines. The first line:

```
myfile: file1.o
```

describes a dependency - in this case the file named **myfile** is dependent on an object file named **file1.o** (the **.o** file extension denotes an object file). Line 3 also has a colon separator and describes the dependencies of the **file1.o** file:

```
file1.o: df1:source/file1.c headers.h
```

The dependencies for **file1.o** are located on drive **df1**: in the directory named **source** for **file1.c** and in the current directory for **headers.h** - **file1.c** would have an include directive for the **headers.h** file.

You should now see a hierarchy of file dependencies for the executable file named **myfile**. Lines 2 and 4 are the action lines - in the case of line 4:

```
lc -b0 -r0 -odf1: df1:source/file1.c
```

this is a standard command line for compilation, with various options being specified.

In line 2, the action is that of linking:

```
blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.li
```

Refer to your compiler manual for further details of the command line options for linking.

7.5 Invoking LMK

LMK can be operated from the Command Line Interpreter (CLI) of AmigaDOS. In addition, **LMK** can also be called under the *Workbench* environment, using the standard *Intuition* icon-mouse actions.

To invoke **LMK**, the simplest method is to create your **lmkfile** using an editor such as the **Lattice Screen Editor (LSE)**, then name this file **lmkfile.** On calling **LMK**, you need only enter the following at the CLI prompt:

```
lmk
```

Your **lmkfile** can also be named **lmkfile.lmk** or **makefile**. The search order of filenames to which **LMK** operates when you invoke LMK without any other flags is:

```
lmkfile
lmkfile.lmk
makefile
```

A file with one of these names must be in your current directory to call **LMK** correctly.

7.5.1 File Naming Options

Depending on your file naming conventions or if you need multiple **lmkfiles** in the same directory, you may wish to name your **makefile** something other than **'lmkfile'** or **'makefile'**. Consider, for example, where you have given your **makefile** the name **'alpha.lmk'**. **LMK** would be invoked with either of the following commands:

```
lmk -f alpha.lmk
lmk -f alpha
```

Either of these command lines signifies to **LMK** that it should take input from the file specified after the **-f** flag rather than from the default filenames.

File name extensions are not generally given to executable files running under the *Workbench* environment. To be called from the *Workbench*, an executable file must have an icon. The graphical data for the icon is held within a file which uses the same name as the executable file, albeit with an **.info** name extension. This has implications for file naming conventions as demonstrated in the following example.

Consider an ASCII text file with the name **readme.txt**. Attaching an icon to this file will result in two files - a file named **readme.txt** and a file named **readme.txt.info**. The former is the text file, the latter is the icon file. The dot or period (.) character is recognized by AmigaDOS as the filename separator. Thus files with name extensions could be construed as unnecessarily confusing.

Note that it is not recommended that you use the second type of command line of **lmk -f alpha** and also have an executable file for **alpha** resident in the current directory. The order of precedence for LMK when using the **-f** flag is to locate a file with the exact name given after the **-f** flag. If this search is unsuccessful, it will search for a file with the name specified with an extension of **.lmk**. If you receive an error message from LMK such as **'Line too long'** or **'Unexpected punctuation'**, you should try renaming your file with an explicit **.lmk** extension, then giving the full name of the file when invoking **LMK**.

When running **LMK** from the *Workbench*, the search order of files is based on the assignment of the device **lmk_files:** (this is done from the CLI using the command line **assign lmk_files: your-device:**). When no assignment has been made, **LMK** searches the current directory for the required files.

7.5.2 Command Syntax

LMK is precise about the syntax it will accept from a makefile. As with any programming or specification language, it is important to follow the syntactic rules in order for **LMK** to understand your requirements.

The target name must:

- Start in the first column.
- Be followed by a colon (:).
- Be followed by the name of any dependencies of the target.

Below the target-dependency line are any action commands needed to build the target from its dependencies. Each of the action lines, if any, must be indented a minimum of one tab or space.

Each sub-target and its dependencies follow the same syntax as the target-

dependency syntax given above. Similarly, the action lines of sub-targets must be indented a minimum of one tab or space.

Comment lines within an **lmkfile** are prefixed with a hash (#) sign. An example **lmkfile** with comment lines is shown below:

```
myfile: file1.o
  blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.lib
# This line is a comment line.

file1.o: df1:source/file1.c headers.h
   lc -b0 -r0 -odf1: df1:source/file1.c
# This line is a another comment line.
```

7.6 Macros

Macros are a method of overcoming potential keyboard entry errors where long character strings are required, as well as saving time by using shorthand to represent a constant string of text. In common with macro usage in the C programming language or with debuggers, a macro invocation is expanded during processing.

Macros in an **lmkfile** can be used for representing directory specifications, header files and flags to the compiler or assembler. Any AmigaDOS command can be encapsulated into a macro.

Some example macros look like this:

```
PASS1 = df1:lc/lc1
FLAGS  = / -b0 -r0 -odf1:
```

The first example is specifying a directory path to the compiler; the second example is specifying compiler option flags.

A macro can also be defined to be the empty string:

```
MNAME =
```

To illustrate how macros are utilized consider our earlier example of an **lmkfile**:

```
FLAGS = -b0 -r0 -odf1:
SOURCE = df1:source/

myfile: file1.o
blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.li

file1.o:$(SOURCE)file1.c headers.h
  lc $(FLAGS)
```

Our original file looked like this:

```
myfile: file1.o
blink FROM LIB:c.o+file1.o TO myfile LIB LIB:lc.lib+LIB:amiga.li

file1.o: df1:source/file1.c headers.h
   lc -b0 -r0 -odf1: df1:source/file1.c
```

The differences between the two files lies in the first two lines and the last line. The first two lines are the macro definitions:

```
FLAGS = -b0 -r0 -odf1:
SOURCE = df1:source/
```

Line 1 defines the flags for the compiler, while line 2 defines the directory path. The last line in the file shows the greatest difference - originally, it looked like this:

```
lc -b0 -r0 -odf1: df1:source/file1.c
```

It now takes the form:

```
lc $(FLAGS) $(SOURCE)file1.c
```

The differences are the inclusion of the dollar sign ($) and the enclosure of the macro name in parentheses. The dollar sign indicates to **LMK** that a macro is to be processed and the name of the macro follows the dollar sign. Note that the macro name must be enclosed in parentheses immediately following the dollar sign.

7.6.1 Macro Rules and Defaults

In common with any usage of macros, **LMK** has certain rules regarding their deployment. These rules are easy to understand and are not excessively restrictive. The previous subsection mentioned the need for the dollar sign ($) to precede the macro name. In addition the macro name must be enclosed in parenthesis - a further rule relates to empty strings. When an empty or undefined macro is used in an **lmkfile**, it will be expanded to the empty string.

Note that if you do not follow closely the **'define-before-use'** rule with macros, and you use a macro which has not yet been defined in the file, it will receive an empty expansion.

Less common but also useful, are the default macros available after a rule of dependency has been processed, but before any associated actions are taken. **LMK** has a total of five default macros built into it. The first two of the five default macros shown below are set after processing each target-dependency line:

$@ Set to the target filename

$? Set to the list of dependencies younger than the target (i.e. have later time stamps). This includes the full directory pathname structure.

$* Set to the prefix of the **<u>first</u>** dependent filename. In other words the filename extension is disregarded, although the full directory pathname structure is included.

$< Set to the dependent filename which caused the actions. Directory pathname structures are ignored

$> Set to the basename of the dependent file which caused the action. This means any filename extensions together with any directory paths are ignored.

An example of the use of these macros looks like this:

```
obj/new/test.o:   src/new/test.c  hdr/test.h
```

The line above has defined the source file as **test.c** located in the directory structure **src/new** together with the header file **test.h** located in the directory **hdr**. Where the source file **test.c** has a time stamp later than our target file **test.o** then the default macros will have the following values:

Macro	**Value**
S@	obj/new/test.o
$?	src/new/test.c
$*	src/new/test
$<	test.c
$>	test

The remaining three default macros are only set when a transformation rule is used. Refer to the following section for details of these rules.

7.7 Summary

The central concepts to **LMK** are:

Targets - Dependencies - Actions

The colon placement determines which is the **target** or source file - any file whose name is positioned to the right of the colon is determined as a **dependency**.

```
target-file1 : dependent-file list
        command-sequence
target-file2 : dependent-file list
        command-sequence
```

Section 8

Advanced LMK

This section is concerned with the more advanced aspects of **LMK** such as multiple and fake targets, command line options, local input files and transformation rules.

8.1 Multiple Targets

It is desirable to have all the actions associated with building a software product in one place. It might also be the case that the product you are building is actually a collection of separate utilities and for the purpose of source maintenance, you may want to keep all bookkeeping type actions in one file. **LMK** allows for the specification of multiple targets within the same **lmkfile** to suit these needs.

For example, the following **lmkfile** could be used for several of the utilities which compose the product you are using:

```
ppp: diff grep
   @rem ppp done

diff:
   cd diff
   lmk -f diff.lmk
```

```
    cd /

grep:
    cd grep
    lmk -f grep.lmk
    cd /
```

In the **lmkfile** shown above, each of the utilities is calling upon other **lmkfiles**. The relevant **lmkfile** for each utility will contain details of the link and compile sequence. Note the use of the **cd /** command to return control back to the originating directory.

8.2 Alternate Targets

When **LMK** is invoked and no target is specified from the command line, **LMK** defaults to building the first target specified. Consider the following:

```
clean:
    delete $(OBJ)files.o
    delete $(OBJ)os.o
    delete $(OBJ)strbpl.o

backup:
    copy files.c to df1:
    copy os.c to df1:
    copy strbpl.c to df1:
```

Here there are two targets, a primary and alternate target named **clean** and **backup** respectively. Note that it is possible to specify that one or more alternate targets be built instead. The notion of alternate targets can also be applied to any of the sub-targets.

The alternate target specified is **backup**, which invokes several AmigaDOS commands to copy object files from our disk. Notice that these actions depend upon a disk being present in drive **df1:** (if the currently logged drive is **df0:**). Under AmigaDOS the **copy** command can be used to copy to or from any directory, including copying to the current drive.

Note that to use the commands within the macro **backup:** you must use the command line:

```
lmk backup
```

8.3 Default Rules

When many of the sub-targets in an **lmkfile** require identical actions such as compilation or copying to another directory, default rules can be specified once, and invoked silently by **LMK**. A default rule informs **LMK** how to make a transformation from a file with one extension to a target or sub-target of a different extension.

There are a variety of ways to use default rules. A default rule can be given explicitly inside an **lmkfile** or it can be placed into a separate file. There are also certain default rules internal to **LMK** which are a part of this implementation.

LMK decides which action to take when it detects a certain file must be made from its dependencies. This action is done on the basis of the following hierarchy:

- Actions specified below the dependency relation.
- Transformation rules describing how to make a file of one extension from one of another distinct extension.
- **.DEFAULT** rules specified in the same **lmkfile** or in the **lmk.def** file.
- Rules taken from an alternate **.def** file.
- Rules taken from a file named **lmk.def** in the current directory.
- Rules internal to this implementation of **LMK**.

Action rules have highest precedence in how **LMK** determines the building of a specific target. These actions are passed on to AmigaDOS for execution and result in the creation of a target on disk or wherever specified.

8.4 Transformation Rules

Transformation rules inform **LMK** how to make a file with a given extension from a dependent with a given extension. For example, a file set has a suffix

of **.c** and you want to use a transformation rule to change the file suffix of **.c** into a **.o** suffix. The name of **.c.o** would be given to the transformation rule in this instance. For example:

```
.c.o:
   LC $*
```

Therefore, when this transformation rule is present, and no explicit command line has been given in your **lmkfile** to override this rule, then the command sequence for the **.c.o** rule is employed. Transformation rules are commonly applied to source code modules and object modules.

Three default macros are set whenever a transformation rule is used:

$* is set to the prefix of the first dependency filename including any directory paths. In other words the filename extension is ignored. For example, if the dependency filename is:

```
df0:alpha/omega.c
```

then the prefix of this filename is:

```
df0:/alpha/omega
```

Notice that the **.c** extension has been dropped.

$< is set to the dependency filename which caused the actions. Thus, if you have dependencies consisting of:

```
df0:alpha/omega.c  df1:beta/delta.h
```

then the dependency filename in this example is **omega.c** and not **delta.h** file.

$> is set to the basename of the dependency which caused the actions. Using the same dependencies as our previous example:

```
df0:alpha/omega.c  df1:beta/delta.h
```

then the dependency filename with this default macro is **omega.**

When there are no actions or transformation rules pertaining to this relation a **.DEFAULT** rule is taken. Often, **.DEFAULT** rules are used for printing out comment lines to the screen during a **LMK** session. For example, the following lines in an **lmkfile**:

```
.DEFAULT:
     ; this target needs remaking $@
alpha.cpp: alpha.clp
```

will cause the message **'this target needs remaking alpha.cpp'** to be printed to your screen during the **LMK** session, provided **alpha.clp** has a later time stamp than **alpha.cpp**. Note that there are no action rules below the target-dependency line, and that there are no known ways to **LMK** to transform a **.clp** file to a file with extension **.cpp**.

An alternate **.def** file can be specified from the command line with the **-b** option. This allows for multiple **.def** files to reside within the same directory. **LMK** looks for a file named **lmk.def** in the current directory. The file named **lmk.def** has been provided on the distribution disk if you are employing other development systems. When you are using the **Lattice AmigaDOS C Compiler**, the rules in **lmk.def** are the same as those internal to **LMK**.

8.5 LMK Internal Rules

Rules internal to this implementation of **LMK** assume you are using the **Lattice AmigaDOS C Compiler**. These rules are taken from the file named **lmk.def** on the distribution disk. You are able to override these rules if desired, by editing the **lmk.def** file.

```
LC = LC:lc
LC1 = LC:lc1
LC1FLAGS = -iINCLUDE: -iINCLUDE:lattice/
LC2 = LC:lc2
AS = ASSEM:assem
LINK = BLINK:blink

.DEFAULT:
   $(LC)  $(LC1FLAGS) $*
```

```
.a.o:
   $(AS)  $* -i Assem5.0:include -o  $*.o

.c.o:
   $(LC)  $(LC1FLAGS)  $*

.h.o:
   delete  $*.o
   $(LC)  $(LC1FLAGS)  $*
```

8.5.1 Special Symbols

There are a set of special symbols which can be attached to actions within an **lmkfile** and provide different resultant actions. These symbols are:

- **-** The minus symbol (-), when prefixed to an action line, will cause **LMK** to continue even if an error return is generated from the action. Normally, when an action returns an error message, **LMK** will halt. It may be desirable to have **LMK** continue after an error return from a specific action. If there are multiple actions within an **lmkfile** to which you want to apply this symbol, then the command line option **-k** will cause **LMK** to continue after all error returns.

- **@** The at symbol (@) when prefixed to an action line provides for silent operation of that command. In other words the command itself will not be echoed to the screen although its resultant actions will be displayed as normal. To have all actions run silently use the **-s** option from the command line.

- **~** The tilde symbol (~) serves as an escape symbol when prefixed to an action line. This may be useful when calling a process whose name begins with something **LMK** would consider a special character. For example, the minus sign (-). The tilde causes the next single character to be escaped. Note that multiple tilde characters may be used on a single action line.

& The ampersand symbol (&) is provided for source code compatibility with previous implementations of **LMK**. It is ignored in the AmigaDOS version.

8.5.2 Example 1

This example demonstrates the use of macros and default rules. In addition to creating a target executable file, the **lmkfile** will also perform and document other actions associated with a software product, such as cleaning up a fragmented disk, creating a production disk, backing up disks and generating program listings. Although this example may seem initially overwhelming, a full dissertation of this file follows the listing.

```
# An example MAKE file for AmigaDOS
GREP = ram:c/
LCHDRS = stdio.h fcntl.h
OBJ = obj/
LCFLAGS = -iINCLUDE: -iINCLUDE:lattice/ -i/grep/  -o$(OBJ)
LC = LC:lc
LIB = LIB:
# The above line is the logical assignment for Lattice_C:lib

.c.o:
   $(LC)  $(LCFLAGS) -cc  $*

files: $(OBJ)files.o  $(OBJ)os.o  $(OBJ)strbpl.o
   LINK:blink FROM $(LIB)c.o+$(OBJ)files.o+$(OBJ)os.o+ \
$(OBJ)strbpl.o TO files LIBRARY $(LIB)lc.lib+$(LIB)amiga.lib

$(OBJ)files.o: files.c  $(GREP)pat.h  $(LCHDRS)

$(OBJ)os.o: os.c

$(OBJ)strbpl.o: strbpl.c
```

8.6 Discussion

There are several groupings of lines in example 1 for **files.lmk** and they are grouped for the purpose of modularity.

The first group of lines in example 1 are macro definitions, that is, lines 2 to 7, shown below.

```
GREP = ram:c/
LCHDRS = stdio.h fcntl.h
OBJ = obj/
LCFLAGS = -iINCLUDE: -iINCLUDE:lattice/ -i/grep/ -o$(OBJ)
LC = LC:lc
LIB = LIB:
```

Lines 2 and 4 are both directory specifications. Line 2 assigns the string for the absolute path name **ram:c** to the macro named **GREP**. Each time the macro **GREP** is used in this **lmkfile**, it will be expanded to the full string it represents. Line 3 assigns a relative path name, namely a subdirectory of the current directory, to the macro named **OBJ**.

Notice that the entire directory structure of your filing system is at your disposal. You can call in header files, source modules, write out object files from anywhere by using the appropriate macros.

Line 3 of the macro definitions assigns to the macro name **LCHDRS** the string **stdio.h fcntl.h**, and will serve as shorthand later if you have multiple files which require these headers.

```
LCFLAGS = -iINCLUDE: -iINCLUDE:lattice/ -i/grep/ -o$(OBJ)
```

Line 5 of the **lmkfile** as shown above holds the string of flags you want to use each time you invoke the compiler from **LMK**. There are several points to note about this line. The most important point is that if you had to enter this line each time it was used, the chances for errors are significant. In addition, a change in the flag settings would entail editing only one line of the **lmkfile**, rather than each line in which the macro is used.

Note, that changing the flags associated with making a specific target will not

affect the time stamps associated with any of your files - hence **LMK** will not be aware that your files need to be rebuilt. To unconditionally make all targets in such a case, the **-u** flag can be used.

Specifically, the compiler flags specify the directory locations of header files with the **-i** option and direct where the resultant object file is to be placed.

```
OBJ = obj/
LCFLAGS = -iINCLUDE: -iINCLUDE:lattice/ -i/grep/  -o$(OBJ)
```

Notice that macros can be nested within other macros. The macro **$(OBJ)** in line 4 will be expanded to the string **obj/** when it is processed. Since **LMK** makes only one pass on an **lmkfile**, any nested macros must be declared before being used in an expression.

Logical name assignments are used in the macro strings. These are known to AmigaDOS, provided they have been defined in your **startup-sequence** file and these are passed on by **LMK** when an action using them is called. **INCLUDE:, LC:** and **LIB:** are logical assignments to specific subdirectories on the **Lattice AmigaDOS C Compiler**.

```
LC = LC:lc
LIB = LIB:
```

The macro defined on line 6 as shown above, represents the string needed to call the compiler driver **lc**, which invokes both phases of the compiler. Line 7 is another directory specification macro, this time using a logical name which has been defined to your system. Notice the comment line which follows these two lines. **LMK** will ignore everything after the hash sign **#**.

```
.c.o:
   $(LC)  $(LCFLAGS) -cc  $*
```

Lines 9 and 10 are interesting since they give a transformation rule. They specify a template of action for converting a file with extension **.c** into one with extension **.o**.

While this transformation is used only once within this **lmkfile**, it can be quite

useful if you have a whole group of files which depend on it. It is also possible to have multiple transformations of the same type in the same file. That is, later on another **.c.o** template can be specified which would apply to all files following it which do not have actions specified, and need a default rule.

A point worth considering in terms of the use of transformation rules is **template matching**. In order for **LMK** to detect the need for a transformation rule, it must see that the first dependency file following the target matches the dependency file specification. In other words, **LMK** will not look at the extensions of the other targets, it will only detect the first, and apply a transformation rule if it matches the template.

```
$(LC)  $(LCFLAGS) -cc  $*
```

A further component to consider about line 10 is the use of the default macro, **$***, which refers to the prefix of the **.c** file. Shown below is line 11 which specifies the dependencies of the target files.

```
files: $(OBJ)files.o  $(OBJ)os.o  $(OBJ)strbpl.o
```

Since line 12 shown below is longer than the standard 80 column screen allows, it has been continued for legibility with the continuation character, the backslash ****.

```
   LINK:blink FROM $(LIB)c.o+$(OBJ)files.o+$(OBJ)os.o+ \
$(OBJ)strbpl.o TO files LIBRARY $(LIB)lc.lib+$(LIB)amiga.lib
```

Thus line 13 is effectively an extension of the preceding line. Line 12 is the final action taken to build the target files from its dependencies.

```
$(OBJ)files.o: files.c  $(GREP)pat.h  $(LCHDRS)
```

Line 14 shown above details how to make the sub-target **files.o**. Notice that a macro can be used anywhere in an **lmkfile**, including the beginning of a target filename.

```
$(OBJ)os.o: os.c
```

The same applies to line 15 which demonstrates you can use additional flags.

```
$(OBJ)strbpl.o: strbpl.c
```

Initially, the line shown above, namely line 16 may look a little peculiar since there are no actions associated with it. On examination of the file extensions of sub-target and dependency, notice that you are trying to transform a file with extension **.c** into a file with extension **.o**. Following the precedence for default rules, you see that there is a transformation rule associated with such an action on lines 12 and 13.

8.7 LMK Options from the Command Line

-a This option will rebuild all targets without regard to time stamps. All targets and sub-targets are rebuilt.

-b file This option will use the filename specified in **file** as the default file, rather than the file named **lmk.def**.

-c This option will create a batch file from the current **LMK** session. The batch file will be passed for execution with the command **execute lmkfile.bat**. The file **lmkfile.bat** will be left in the current directory giving a history of the actions and responses of the **LMK** session.

-d This option will print out detailed debugging information about the processing of the **lmkfile**.

-e This option will erase any out-of-date targets before re-making them.

-f file This option will use the filename specified in **file** as the input **lmkfile**.

-h This option will print out help information and exit.

-i This option will ignore error returns from actions.

-k This option will ignore error returns from actions passed to AmigaDOS. In addition, this option will ignore error

returns stemming from **LMK** not knowing how to make certain targets.

-n This option will display the actions **LMK** would have taken onto the screen, but will not execute these actions.

-p This option will print out target descriptions and expanded macros.

-q This option will query if the **target** file is updated. Print a 1 if the target is currently up to date or a 0 if it is not. **LMK** will not take any of the associated actions.

-s This option will not echo actions to the screen before executing them.

-t This option will **Touch** the target files by updating them with the current system time. None of the actions usually associated with making these targets will be performed.

-u This option will rebuild unconditionally all targets without regard to time stamps. This will force everything (all targets and subtargets) to be rebuilt.

-x This option is for UNIX compatibility. If this option is used, **LMK** attempts to detect any features of the **lmkfile** which would prevent it from interfacing correctly with the UNIX **make** utility.

Examples of such features are:

- Use a local input file.
- Define an action line without an initial tab character.
- Specify multiple target files on a single target-dependency line.

In each case, **LMK** will display an informative error message describing the incompatibility. This option is intended for those users who wish to employ the same **lmkfile** or **lmkfile** on a variety of operating systems.

Both the **-k** and **-i** actions should be used with caution, as **LMK** will continue

its actions. Note that the **-k** option incorporates all of the functionality of the **-i** option.

To embed a literal dollar sign ($) in an **lmkfile**, for example within a filename, you should use a second dollar sign to escape the first sign. This will prevent **LMK** from interpreting the dollar sign as a macro invocation. At times, it may be wise to defeat some of the actions **LMK** would otherwise take. Although **LMK** is an intelligent tool, it has no knowledge of your source files other than through the dependencies you have given in your **lmkfile**.

8.7.1 Macro Command Line Override

It is possible to override the definition of a macro from the command line of **LMK**. This provides for an efficient method of testing new actions or making quick revisions without making a permanent change to our **lmkfile**.

Consider the previous example - suppose you wish to use a different assembler, an assembler which generates 68020 code rather than 68000 code. You could invoke **LMK** to use your 68020 assembler **asm20** by using the following command:

```
lmk -f grep.lmk ASM=asm20
```

and hence provide **LMK** with a new definition of the macro **ASM**. Note that no spaces are allowed between the macro name, the equals sign (=) and the macro definition. This example is trivial in that it would require the new assembler to have the same syntax as the previous one.

8.7.2 Alternative Definition Files

To use an alternate **.def** file, the following command line syntax is used:

```
lmk -b other.def
```

This will signify to **LMK** that this file should be used in place of **lmk.def**, in situations where these rules are called for.

8.8 Local Input Files

A local input file provides for a way of creating a temporary disk file consisting of actions specified within an **lmkfile**. The temporary file is named **temp_lmk.tmp** by default, and is automatically deleted before the end of the **LMK** session.

Local input files provide a method of keeping actions associated with making a target in the same **lmkfile** rather than in a separate file on disk. Any macros defined in the **lmkfile** can be used within a local input file.

Commands to appear within the temporary file are enclosed between the two delimiter characters of < and <.

A template for construction is:

```
command <[preface]<[!] [ (filename) ]
    stmt1
    stmtN
    <
```

where:

command	The action to be handed to AmigaDOS.
preface	The symbol which is to precede the local input filename when it is given to AmigaDOS.
!	The instruction to **LMK** to create a local input file.
filename	The optional filename given to the local input file created.
stmt1-N	The statements within the local input file.

If the optional exclamation point **!** is used after the second <,then the name of the local input file will <u>**not**</u> appear on the generated command line. Note that the file **preface** may contain spaces. The local input filename, must be enclosed in parentheses and no spaces may occur either before or after this filename.

8.8.1 Example 2

The final example illustrates a feature of **LMK** - the use of local input files.

This example is an **lmkfile** creating the target **GREP** a UNIX derived utility for finding patterns in files. **GREP** is composed mainly of C source files, with the addition of one assembly language module. Since the **GREP** utility relies heavily on command line input, you are using a local version of the **Lattice AmigaDOS C Compiler** startup module named **c.a**.

```
# GREP example for AmigaDOS
LCHDRS = grep.h pat.h
OBJ = obj/
FLAGS = -iINCLUDE: -iINCLUDE:lattice/ -o$(OBJ)
LC = LC:lc
ASM = LC:asm

grep: $(OBJ)grep.o  $(OBJ)c.o  $(OBJ)_main.o  $(OBJ)re_gen.o
   BLINK:blink <WITH <
FROM $(OBJ)c.o+$(OBJ)grep.o+$(OBJ)_main.o+$(OBJ)re_gen.o
TO grep
LIBRARY LIB:lc.lib+LIB:amiga.lib
<

$(OBJ)grep.o: grep.c $(LCHDRS)
   $(LC) $(FLAGS) -dGREP grep

$(OBJ)c.o: c.a
   $(ASM) c.a -i Include:Assembler _Includes -o $(OBJ)c.o

$(OBJ)_main.o: _main.c
   $(LC)  $(FLAGS) -dGREP _main

$(OBJ)re_gen.o: re_gen.c $(LCHDRS)
   $(LC)  $(FLAGS) -dGREP re_gen
```

8.9 Discussion

The **lmkfile** to create the target **grep** is shown above. The macros are identical to those in the first example in this section of the manual with the addition of the macro **ASM.** This is defined to be the string necessary to invoke the **Lattice 68000 Macro Assembler** located in the subdirectory named **ASSEM** - the logical assignment which should be in your **startup-sequence** file.

```
grep: $(OBJ)grep.o  $(OBJ)c.o  $(OBJ)_main.o  $(OBJ)re_gen.o
grep: $(OBJ)re_match.o  $(OBJ)os.o
```

Line 6 gives the target **grep**, and its dependencies. If there are too many dependencies to fit on one line, continuation is specified by retyping the target filename and a colon (:) on all subsequent lines.

```
BLINK:blink <WITH <
```

Line 7 begins with a command to invoke the linker. The first less-than symbol (<) on the line informs **LMK** that the following lines, up to the terminating delimiter (<), will be taken as a local input file.

Between the first delimiter and the second delimiter on line 7 is the command you want to appear after the link command but before the input filename. The filename given to the local input file is by default **temp_lmk.tmp**, therefore ensure you do not have a file of the same name resident in the current directory.

The command passed to AmigaDOS is:

```
BLINK:blink WITH temp_lmk.tmp

FROM $(OBJ)c.o $(OBJ)grep.o $(OBJ)_main.o
FROM $(OBJ)re_gen.o $(OBJ)re_match.o $(OBJ)os.o
TO grep
LIBRARY LIB:lc.lib+LIB:amiga.lib
```

Lines 8 to 10 are the commands which will be placed into the temporary file. Note that line 8 carries over in the fragment of code shown above. A **WITH**

file can be used when calling the linker - the linker will take its parameters from the file instead of the command line. Refer to the **Lattice AmigaDOS C Compiler** manual for full details about the linker.

Line 11 terminates the local input file with a closing delimiter (<).

Notice the unique syntax required for the assembler. A space is required between the **-i** and its arguments and also for the **-o** flag and its argument. Refer to the **Lattice AmigaDOS C Compiler** manual for full details about the assembler.

8.10 Fake Targets

LMK provides support for a number of *fake* targets. In other words, an expression which appears to be a target in an **lmkfile** is actually used to control the operation of **LMK**. Fake target names must always begin with a period (.) and be capitalized. The degree of control on the behavior of **LMK** is usually that of inserting lines into the **lmkfile**. There are five options available for *fake* targets:

- .DEFAULT
- .IGNORE
- .ONERROR
- .SET
- .SILENT

The results of each *fake* targets are tabulated below:

Fake Target	Result
.DEFAULT	Specify default rules
.IGNORE	Same as -i option
.ONERROR	Specify error response
.SET	Set environmental variable
.SILENT	Same as -s option

The following part of this subsection provides a description of these options for *fake* targets.

.DEFAULT This option is used to specify an individual action or a set of actions to be executed based on two parameters. These parameters are:

- There are no explicit instructions.
- There is no default transformation rule for the given target/dependency pair.

The command line syntax used for **.DEFAULT** is:

```
.DEFAULT:
      action-1
      action-
      .....
      .....
      action-N
```

Consider a dependency line such as:

```
alpha.x:   alpha.y
```

where this dependency line had no explicit action and no transformation rule with **.y.x** as the target. In the presence of the **.DEFAULT** rule, the various actions shown above **action-1**, **action-2** and **action-N** would be executed.

.IGNORE This option is equivalent to having called **LMK** with the **-i** flag. This means that any error returns from actions will be ignored. The command line syntax used for **.IGNORE** is:

```
.IGNORE:
```

.ONERROR This option is used to specify an action or set of actions to be performed when **LMK** detects an error condition. For example, consider where **LMK** is to be called recursively from within a high-level **lmkfile**:

```
product: alpha beta
    rem product finished

alpha:
    cd dir1
    lmk -f alpha.lmk

beta:
    cd dir2
    lmk -f beta.lmk
```

When one of the lower level invocations of **LMK** exists with an error, you will be left in one of the subdirectories - either **dir1** or **dir2**. In order to finish in a higher directory level you would add these lines to your

lmkfile:

```
.ONERROR:
    cd /
```

Alternatively, you may want to delete certain temporary files or transfer them elsewhere if an error condition is detected. Either of these actions can be directed by means of the **.ONERROR** option.

.SET

This option allows you to adjust the values of logical name assignments from within an **lmkfile**. When the logical name is set in this way, it retains the specified value. Note that this also applies to any secondary processes spawned from the main process.

The command line syntax used for **.SET** is:

```
.SET:     name = value
```

where **name** is the logical name you wish to set, and **value** represents the value to be assigned. The contents of **value** may be any character, including white space.

However, the leading and trailing white space characters will be stripped off if these are present.

The **.SET** option is useful in defining the logical name **INCLUDE**. For example:

```
.SET:    INCLUDE: = df0:lc/examples
```

.SILENT This option is equivalent to having called **LMK** with the **-s** flag. This means that during **LMK** operation, the action about to be executed will not be echoed to the standard output device, usually the screen. The command line syntax used for **.SILENT** is:

```
.SILENT:
```

> NOTE: For the **.DEFAULT** fake target, the actions will **always** be executed - even if explicit actions have been specified. The default actions will be executed first, and then the explicit actions will be executed.

Section 9

Lattice Profiler

9.1 Profile and Report Generator

The Lattice profiler **lprof** and associated report generator **lstat** allow you to profile the execution of your application code. The profiler records the amount of time that is expended in each routine that is entered during the execution of your application. The report generator, **lstat**, reports the times to you in a statistical report. This allows you to do a performance analysis on your program, identifying "hot-spots" so that intelligent improvements can be made.

When you use a profiler, you should have some objectives in mind, including the application re-design steps you may want to carry out. A large percentage of time spent in some program module may indicate that:

- the module should be redesigned,
- the module should be incorporated into the calling routines thus eliminating calling overhead,
- the routine should be re-coded in assembly language, or
- the routine needs to be modified as part of a major re-structuring of the application.

It may happen that most of the work being done involves the routine in question, and that the percentage of time taken is reasonable.

Considerations such as those just mentioned should be done as you collect and analyze data from the profiler.

9.2 LPROF Command

The **lprof** command is used to gather information about where a program is spending most of its execution time. In order to gather statistics on a given command, prefix the command with the **lprof** command.

```
lprof command
```

> Note: If the command you wish is not in the current directory, you must fully specify the command location even *if it is in the path*. Also, any redirection operators which were immediately after the command must immediately follow the **lprof** command.

After the profiler loads the command, it will execute the command and gather statistics by examining the current program counter at given intervals. You can verify that the profiling is in progress by the power light fluctuating in intensity. When the program run completes, the **lprof** command will create the file **prof.out** in the current directory. You can then use the **lstat** command to obtain a statistical report on the execution profile.

For instance, you may be interested in analyzing a simple program called **foo** which takes no arguments. You can gather statistics on it by entering the command:

```
lprof foo
```

For a program that normally takes command-line arguments, you can add them after the command name as in:

```
lprof mycmd -opt 1 -flag
```

Assume the program needs to have output redirected as in:

```
myprog >outfile -opt 2 -doit
```

If you intend to run the profiler, you would need to move the redirection operators after the **lprof** command as in:

```
lprof >outfile myprog -opt 2 -doit
```

If the program were in the C: directory or another directory automatically searched by the CLI, you would need to give the full path name as in:

```
lprof C:mynewcmd -opt 2
```

9.3 LSTAT Command

The **lstat** command is used to analyze a profile statistics file created by the **lprof** command. It prints the statistics gathered by the **lprof** command.

The **lstat** command is invoked by:

```
lstat [>output] [options] program profile
```

The **program** field is required and indicates the name of the executable for which the statistics were gathered. **lstat** uses this to obtain all the debugging and symbol information for its report. The **profile** field is optional and is used to override the default of **prof.out** which is written by the **lprof** command. You will need to use this if the file is in a different directory, or you have renamed the file.

The **>output** field is optional and redirects **lstat's** output from the screen to an output device or file. Most programmers use **lstat** by redirecting its output to a file and then printing the file. See the examples which follow.

The **options** field need not be present. **lstat** accepts the following options

-z This option instructs **lstat** to display statistics for all subroutines even if they were not encountered in profiling. The default is to suppress all entries that have a 0 hit count.

-f This option instructs **lstat** to display full statistics for each subroutine indicating the line numbers within a module that were hit by profiling. By default, only summary information about the subroutine is printed.

-t=n This option allows pruning of the information presented. By default, **lstat** will print all subroutines which had even a single hit. If you specify **n** it will only display those which have at least **n** or more hits.

> Note that this command assumes that you have compiled with at least the **-d1** debugging option from **lc1** in order to be able to associate the code with the given line. In the absence of this information, the **lstat** command will only report statistics on a subroutine by subroutine basis.

9.3.1 Examples

These examples assume that the file **lprof.out** is in the current directory. It is created by running the command:

```
lprof testprog
```

To simply view the basic statistics about the run, we can give the command:

```
1> lstat testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                       Most hits -  1.2121% (8) on line 80
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                       Most hits -  1.2121% (8) on line 132
```

```
                      ...
  0.1515% 100.0000%  1ca18       1  CXM22
```

We can include statistics on the additional routines that did not get encountered by the profiler with the **-z** option. Note the extra routines at the end of the following example have 0 hits and account for 0.0000% of the execution time but now appear in the output as a result of using the **-z** option.

```
1> lstat -z testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                       Most hits -  1.2121% (8) on line 80
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                       Most hits -  1.2121% (8) on line 132
                      ...
  0.0000% 100.0000%  1ca60        0  strcat
  0.0000% 100.0000%  1ca78        0  strncpy
```

More information about the individual lines in a subroutine may be obtained with the **-f** option. For each routine that has line number information, it will indicate the percentage of total execution that was spent on that line in the subroutine. Note that subroutines without line number information will not show any additional information.

```
1> lstat -f testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                          0.3030% (2) on line 64
                                          1.0606% (7) on line 68
                                          0.7576% (5) on line 69
                                          1.2121% (8) on line 80
                                          0.3030% (2) on line 90
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                          0.1515% (1) on line 117
                                          0.1515% (1) on line 129
                                          0.4545% (3) on line 131
                                          1.2121% (8) on line 132
                                          0.1515% (1) on line 133
                                          0.1515% (1) on line 138
                 ...
```

```
 0.1515% 100.0000%  1ca18       1  CXM22
```

With a large program, the amount of information presented by the profiler can be overwhelming. In attempting to improve performance of such a program, it is beneficial to limit the amount of information that the **lstat** program presents. The **-t** option can be used to set a threshold below which no information will be reported. With this option, the final total will not be 100%.

```
1> lstat -f -t=2 testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                        0.3030% (2) on line 64
                                        1.0606% (7) on line 68
                                        0.7576% (5) on line 69
                                        1.2121% (8) on line 80
                                        0.3030% (2) on line 90
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                        0.1515% (1) on line 117
                                        0.1515% (1) on line 129
                                        0.4545% (3) on line 131
                                        1.2121% (8) on line 132
                                        0.1515% (1) on line 133
                                        0.1515% (1) on line 138
                    ...
  0.3030%  98.3333%  1ca00        2   xcovf
```

Section 10

SPLAT

SPLAT is a stream or line-based editor for files similar to the AmigaDOS system editor named **Edit**. However, **SPLAT** is both easier to use and faster in operation than the system editor. Given a pattern, a substitution string, and a set of files, **SPLAT** replaces all occurrences of the pattern with the string in each of the files. The original version of the file is left untouched, unless the **-o** option is used. **SPLAT** will either place the result of its substitutions in a temporary file, or within a specified directory with the same root filename, but with an **.$$$** extension.

10.1 SPLAT Command

SPLAT is especially useful when fairly simple substitutions need to be made in a number of source files. For example, consider a situation where you wanted to replace all occurrences of unsigned character declarations with character declarations in each of your C language source files.

The **SPLAT** command to do this would be:

```
splat "unsigned char" char #?.c
```

The effect of this command is to replace all occurrences of the string "unsigned char" with "char" within all ".c" files of the current directory."

SPLAT is more efficient than a text editor since:

- There is no need to read the entire file into memory while **SPLAT** does its substitutions.
- **SPLAT** can be applied once to several files rather than making the same repititious change to a number of files one at a time.
- It will not modify the original version of the file it is working on.

10.2 Using SPLAT

SPLAT is invoked in the following way:

```
splat [-s] [-o | -dPREFIX] [-v] pattern string files
```

where:

-s This option forces **SPLAT** to announce the name of each file as it begins its work on the file and to announce that no substitutions have been performed on a file if it was unable to find the specified pattern in the file. Without the **-s** flag **SPLAT** does its work silently.

-o This option tells **SPLAT** to overwrite the original version of the file with the new version.

-dPREFIX This option requires you to specify a directory prefix indicating the directory into which **SPLAT** will place the new files - leaving the original versions untouched.

-v This option means **verbose** and causes **SPLAT** to display all lines in which substitutions are made to standard output. No files are created when using this option, and it is intended only to give you a quick preview of the changes **SPLAT** will make when it is creating modified files.

pattern This option refers to a regular expression pattern as described in the section dealing with **GREP**.

string This option designates the string you desire to be substituted for the pattern.

Note that **SPLAT** uses the same technique as **GREP** in performing its substitutions. Characters in the substitution string which are special to **GREP**, such as the ampersand (&), are also special to **SPLAT** in the same way. Just like **GREP**, **SPLAT** treats the following characters in the pattern argument as special:

$] . [

Refer to the section on **GREP** for a complete description of these special characters.

Only one of the **-o** or **-d** options may be used since they are incompatible with one another. Thus, the command:

```
splat -d/backup/ int INT #?.headers #?.source
```

causes **int** to be replaced with **INT** in all files with **.headers** and **.source** extensions, and the new versions of these files to be placed in the directory named **/backup**. Note that this command will replace such expressions as **printf** with **prINTf** - which may not be the desired result. Similarly, the command:

```
splat -dbackup/ int INT #?.headers #?.source
```

will perform the same substitutions, but it will place the new versions of the files in the subdirectory **backup/** of the current directory.

If the specified directory does not exist, **SPLAT** attempts to create it. When the attempt to create the directory fails, an error message will be displayed on the screen.

10.3 Examples

In the absence of the **-o** option **SPLAT** will not overwrite the original versions of your files. Therefore, it is quite safe to experiment with **SPLAT**. Such experimentation is often wise since an erroneous specification of the regular expression pattern may result in undesired changes.

For example, we wish to go through each of our source files and substitute **INT** for **int** in every declaration of an integer identifier. However, we do not want our **printf** calls to become **prINTf**. If we know that in each such integer declaration, the expression **int** is preceded and succeeded by a space, then the command to use is:

```
splat -dbackup/" int "" INT " #?.source
```

Recall that in common with **GREP**, a pattern containing white space must be enclosed in double quotes.

For example, on each line of the file **comp.bat** where the expression **lc1** occurs we wish to append the expression **-cc** at the end of the line. Then the command:

```
splat -dbak/ "lc1.*$" "& -cc" comp.bat
```

will perform this objective. The pattern **lc1.*$** will match a string beginning with **lc1** followed by zero or more of any character, and ending with the end of the line. The substitution string **& -cc** will expand the matched string by a space followed by **-cc**.

It is also possible to insert lines in a file with **SPLAT** or to break single lines into multiple lines. For example, if the expression:

```
parms(s);usage();exit(1);
```

occurs on a single line of the file **test.c** and you want any function calls to **usage()** and **exit()** to be positioned on separate lines and indented one tab stop, you would use the command:

```
splat parms(s);usage();exit(1) parms(s);ntusage();ntexit(1);nttext.c
```

to insert the newline and tab characters.

Once you have gained familiarity with **SPLAT**, you will see that complex substitutions may be accomplished with a wide-ranging choice of the regular expression pattern and substitution string.

Section 11

Traceback Utility

The traceback utility (**tb**) is used to debug those situations where your application program is causing an abend, or abnormal termination. The information provided in the traceback utility is the run-time stack, the registers at the time of termination, the memory, application data, and the environment. This information is usually sufficient to enable you to debug the application with this problem.

11.0.1 Command Details

The traceback utility is invoked by:

```
tb [-<options>] [[tbfile] file]
```

This command is used to process a **Snapshot.TB** file produced by a program linked with **catch.o**. This file is a standard IFF format file that contains information about the program abend and environment at the time of the exception.

If given, **options** must one or more of the following characters in any order:

-l This will dump all sections present in the traceback file.

-x Symbol Xref. This dumps the location of all symbols encountered in the program.

-s Stack. This dumps the contents of the entire stack at the time of the snapshot.

-r Registers. This displays the current contents of registers at the time of the snapshot.

-v Environment. This is the default to display where the program was executing and the call back chain.

-m Memory. If present in the snapshot file, this will display the amount of memory available at the time of the snapshot.

-u User data. If present in the snapshot file, this will display any program generated user data section.

If no options are given, **tb** will default to printing out the current stack frame call back to indicate the current code location.

The optional parameter **file** indicates from which to read the debugging information. If not given, the **tb** command will use the program specified in the snapshot file.

> Note that a file option is most useful when you use two images: one with full debugging information and another without debugging information that you normally run. In this way, you get the benefits of faster loading of the non-debug version yet still have access to the traceback information if it does crash.

The **tbfile** parameter is used to override the default of **Snaptshot.TB** in the current directory. If you give this option, you MUST also specify a program name.

11.0.2 Examples

Assume that you have a program *testit* linked with **catch.o** that has created a file **Snapshot.TB** in the current directory. We run the **tb** command to produce a traceback:

```
1> tb
Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110  211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde =  foo line 5
   called from 22156a = main line 11
   called from 221c60 =  main + 704
   called from 2210ca = hunk 0 + $152
```

To obtain a full dump of all information, we can use the **-l** option:

```
1> tb
TraceDump 0.88 Copyright (c) 1988 The Software Distillery

TraceDump Utility: catch.o; Version 1, Revision 0
Processor type: 68000
VBlankFreq 60, PowerSupFreq 60

Symbols for hunk 0
               foo = 22153c                main = 22155c

Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110  211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde =  foo line 5
   called from 22156a = main line 11
   called from 221c60 =  main + 704
   called from 2210ca = hunk 0 + $152

Registers:
D0=00221cde D1=00000000 D2=00000001 D3=00002718
D4=00000001 D5=0000002b D6=0000003b D7=00000000
A0=00221f54 A1=00243fcc A2=0024460a A3=00225c68
A4=00211268 A5=002445be A6=002033c8 A7=002445ae
PC 221cde   C=0 V=0 Z=1 N=0 X=0

stack top: 244640, stack pointer: 2445ae, stack length: $94
entire stack, size = $94 bytes
2445AE: 000003ED 00221558 0000002B 00000000 : ...m.".X...+....
2445BE: 002445CA 0022156A 00225C68 002445F6 : .$EJ.".j."h.$E.
2445CE: 00221C60 00000001 00211620 00225CC4 : .".`.....!. ."
2445DE: 00226030 00225C68 83D10000 00000021 : ."`0."h.Q.....!
2445EE: 13230021 00211620 00244640 002210CA : .#.!.!. .$F@.".J
2445FE: 00244602 74657374 6974000A 00000022 : .$F.testit....."
24460E: 60300000 27100000 27180000 00010000 : `0..'...'.......
24461E: 002B0000 003B0022 60300022 61300020 : .+...;."`0."a0.
```

```
24462E: 35300022 0F740024 464400FF 425800FF : 50.".t.$FD..BX..
24463E: 424C0022                            : BL."
```

The **tb** utility can present each individual piece of the information based on additional option letters. In the above dump, you can examine the sections:

(1) Xref section **-x** option

(2) Environment section **-e** option

(3) Register section **-r** option

(4) Stack section **-s** option

Section 12

TOUCH

TOUCH is a utility which adjusts the time and date stamp on specified files to the current system time and date. This allows you to change the file time stamp to any date and time within system limits. In the case of the current version of AmigaDOS, this means a time stamp within the range 2 January 1978 - 31 December 2045.

12.1 Using TOUCH

The basic command line syntax for **TOUCH** is:

```
touch [-m] file1 file2 ...
```

Note that there is only one option for **TOUCH** which refers to a wildcard. The *-m* option is employed when you are using the MS-DOS wildcard convention of the asterisk. For example, consider the listing of a directory which gives the following:

```
alpha1          5304  rwed   31-Jul-86  10:03:45
amiga            Dir  rwed   05-Aug-86  14:34:12
alpha3          1904  rwed   31-Jul-86  10:04:22
beta3            777  rwed   07-Aug-86  02:22:09
```

```
beta1            6672  rwed   31-Jul-86  10:04:35
gamma2           1392  rwed   31-Jul-86  10:04:58
gamma1           3280  rwed   31-Jul-86  10:05:16
assembler         Dir  rwed   05-Dec-87  09:56:33
alpha2           4869  rwed   31-Jul-86  10:05:42
beta2            6432  rwed   12-Nov-87  16:23:03
```

If system date is 08-Jan-88 and the system time is 00:00:00 (midnight), entering the command:

```
touch -m b*
```

will result in a directory listing of:

```
alpha1           5304  rwed   31-Jul-86  10:03:45
amiga             Dir  rwed   05-Aug-86  14:34:12
alpha3           1904  rwed   31-Jul-86  10:04:22
beta3             777  rwed   08-Jan-88  00:00:01
beta1            6672  rwed   08-Jan-88  00:00:02
gamma2           1392  rwed   31-Jul-86  10:04:58
gamma1           3280  rwed   31-Jul-86  10:05:16
assembler         Dir  rwed   05-Dec-87  09:56:33
alpha2           4869  rwed   31-Jul-86  10:05:42
beta2            6432  rwed   08-Jan-88  00:00:03
```

Note that the files **beta1**, **beta2** and **beta3** have been updated. The command:

```
touch a#?
```

specifies the use of AmigaDOS wildcards **#?** and results in a directory listing of:

```
alpha1           5304  rwed   08-Jan-88  00:00:01
amiga             Dir  rwed   05-Aug-86  14:34:12
alpha3           1904  rwed   08-Jan-88  00:00:02
beta3             777  rwed   07-Aug-86  02:22:09
beta1            6672  rwed   31-Jul-86  10:04:35
gamma2           1392  rwed   31-Jul-86  10:04:58
gamma1           3280  rwed   31-Jul-86  10:05:16
assembler         Dir  rwed   05-Dec-87  09:56:33
alpha2           4869  rwed   08-Jan-88  00:00:03
```

```
beta2             6432  rwed    12-Nov-87  16:23:03
```

The files **alpha1**, **alpha2** and **alpha3** have been updated. Note that directories are unchanged by **TOUCH**, thus the time stamps on **amiga** and **assembler** remain unchanged.

To amend the time stamps on specific files, the command line:

```
touch gamma1 gamma2
```

would result in a directory listing of:

```
alpha1            5304  rwed    31-Jul-86  10:03:45
amiga              Dir  rwed    05-Aug-86  14:34:12
alpha3            1904  rwed    31-Jul-86  10:04:22
beta3              777  rwed    07-Aug-86  02:22:09
beta1             6672  rwed    31-Jul-86  10:04:35
gamma2            1392  rwed    08-Jan-88  00:00:01
gamma1            3280  rwed    08-Jan-88  00:00:02
assembler          Dir  rwed    05-Dec-87  09:56:33
alpha2            4869  rwed    31-Jul-86  10:05:42
beta2             6432  rwed    12-Nov-87  16:23:03
```

Section 13

WC

WC (Word Count) is a companion utility to **DIFF** and will tabulate the number of characters, words and lines within a file. The concept of a *word* as used by WC is an intuitive one, *word* ends when whitespace (a space, tab, carriage return or linefeed) is encountered. In this context, a *word* is a sequence of printable characters.

A *line* ends with a newline character. WC operates in *translated mode* which means a file which contains a combination of a newline character followed by a carriage return character, is reduced to a single linefeed character.

13.1 Using WC

The command line syntax for WC is:

```
wc [-p] [-s] file
```

WC will respond by displaying counts of the number of characters, words, and lines in the specified file. The options are described below.

WC provides a convenient way of obtaining some basic statistics on a file. It

can be helpful in using **DIFF** with the **-l** option. In this case, **WC** can detail how large the line number tables for **DIFF** should be.

13.2 Options

WC has two options:

-p — This option will filter any printable characters. This action is identical to the **-p** option of **GREP** and **DIFF** and removes non-printable characters from the input stream.

-s file — This option calculates a checksum for the filename specified in **file**. In addition, this option also provides information concerning the number of characters, words and lines in the specified file. The checksum is calculated in two parts. The first part is simply the number of printable ASCII characters in the file. The second part is computed with the help of an internal table which matches printable ASCII characters uniquely to numbers - the exact correspondence is unimportant. Each time a printable ASCII character (excluding whitespace) is encountered in the file, its numeric representative from the table is added to the sum. The final sum is the second part of the checksum reported by **WC**.

13.3 Checksum Details

The checksum consists of the two parts, and is displayed in the following format:

```
(number of printable ASCII characters)   (checksum)
```

This checksum provided by **WC** can be useful in one of two circumstances:

- It provides a quick check on whether two files are identical in terms of content. If their checksums differ, then they are **not** identical.

- It can be used in the context of uploading or downloading a file from one machine to another to ensure that no inadvertent character translation has taken place.

This can be troublesome when uploading a text file from the Amiga to an IBM® mainframe computer which uses the EBCDIC character set, or downloading from such a mainframe to the Amiga. Since the character representation table of **WC** is independent of the character set used, and since it ignores whitespace, then the net result of running **WC** on a file on the Amiga and on the same file uploaded to an IBM mainframe should be that the checksums are identical.

In order to carry this out you must have **WC** on both the host and target machines. For this reason we have provided the source code of **WC**. You only need to transfer it to the mainframe and compile **WC** there. It is sufficiently small to be directly entered from a terminal keyboard. Many hours of needless debugging can be avoided by comparing checksums after file transfer.

Appendix A

GREP Libraries

NAME

re_gen Generate a regular expression pattern

SYNOPSIS

```
#include "pat.h"
pat = re_gen(str)
PATTERN pat;  pattern generated
char *str;    character string representation of the pattern
```

DESCRIPTION

The function **re_gen** translates the string representation **str** of a pattern into an internal representation **pat** described in **pat.h**. This **PATTERN** may then be used by the functions **re_match( )** and **are_match( )** to match against a string.

RETURNS

The **PATTERN**, if successful; otherwise **NULL**

SEE

grep, re_match( )

NAME

re_match Regular expression match

SYNOPSIS

```
#include "pat.h"
found = re_match(str,pat)
int found;
int index;                index to anchor match
PATTERN pat;              regular expression pattern
char *str;                string being scanned
```

DESCRIPTION

The function **fe_match** scans the specified string to determine if it contains an occurrence of the given pattern. The pattern is an internal representation of a regular expression (RE), usually obtained via a call to the function **re_gen**( ). Note that you must **#include** the file **pat.h** in order to declare any variable of type **PATTERN**.

RETURNS

found = the index of the last character in the first sub-string of **str** which matches **pat**, if there is one.

found = -1 if there is no match.

NAME

are_match Anchored regular expression match

SYNOPSIS

```
#include "pat.h"
found = are_match(str,index,pat)
int found;
int index;              index to anchor match
PATTERN pat;            regular expression pattern
char *str;              string being scanned
```

DESCRIPTION

The function **are_match** scans the specified string to determine if it contains an occurrence of the given pattern. The pattern is an internal representation of a regular expression (RE), usually obtained via a call to the function **re_gen( )**. Note that you must **#include** the file **pat.h** in order to declare any variable of type **PATTERN**.

RETURNS

found = the index of the last character in the first sub-string of **str** which matches **pat**, if there is one.

found = -1 if there is no match.

NAME

re_smatch Regular expression match of strings

SYNOPSIS

```
#include "pat.h"
found = re_smatch(str1,str2)
int found;
int index;           index to anchor match
char *str1;          pattern string
char *str2;          string being scanned
```

DESCRIPTION

The function **re_smatch** scans the specified string **str2** to determine if it contains an occurrence of the given pattern **str1**. The string **str2** denotes a pattern as characterized in previous sections. **are_smatch( )** is like **re_smatch( )** except that it performs an anchored search from index. These functions are similar to **re_match( )** and **are_match( )** with the exception that the pattern specified in **re_match( )** and **are_match( )** is the internal representation of a pattern produced by **re_gen( )**.

SEE

re_match, **are_match**, and **are_smatch**

RETURNS

found = the index of the last character in the first sub-string of **str2** which matches **str1**, if there is one.

found = -1 if there is no match.

NAME

are_smatch Anchored regular expression match of strings

SYNOPSIS

```
#include "pat.h"
found = are_smatch(index,str1,str2)
int found;
int index;            index to anchor match
char *str1;           pattern string
char *str2;           string being scanned
```

DESCRIPTION

The function **are_smatch** scans the specified string **str2** to determine if it contains an occurrence of the given pattern **str1**. The string **str2** denotes a pattern as characterized in previous sections. **are_smatch**( ) is like **re_smatch**( ) except that it performs an anchored search from index.

SEE

re_smatch, **re_match**, and **are_match**

RETURNS

found = the index of the last character in the first sub-string of **str2** which matches **str1**, if there is one.

found = -1 if there is no match.

Appendix B

File Matching

This appendix discusses the implications of file matching and its impact upon **DIFF**. There are several different designs available for a file comparator program. The one used here is described in *A Technique for Isolating Differences Between Files* - P. Heckel, Communications of the ACM, April 1978, pp.264-268.

B.1 Theory of File Matching

The matching algorithm described by Heckel is simple, fast and effective. Briefly, it works as follows:

- Identify those lines which occur only once in each file. Assume that these lines match one another.
- Sweep downward into the first file. Each time you come to a line **L** which is already matched to the line $\mathbf{L^1}$ in the second file, examine the next line. This second line is referred to as **L+1**.
- If **L+1** is the same as the corresponding line $\mathbf{L^1+1}$ in the second file, then match those lines as well.

- Once this downward sweep is completed, sweep upward in the first file attempting to match in the same way - except you look at **L-1** and **L^1-1** when finding a matched line **L**.

There are some problems with this algorithm which a knowledgeable user may wish to consider. Notice that the effect of the matching algorithm is to match blocks of text to each other in the two files. However, as just described, the algorithm frequently will cross match single lines or two blocks of lines. That is, lines **L** and **L+k** may be matched to lines **L^1** and **L^1-j**. Consequently, after the process described above is completed, the matches must be **untangled** in order to generate a coherent report of the differences.

The concept employed is to examine two cross-matched blocks and then **unmatch** the smaller of the blocks. This is an attempt to satisfy the notion that the differences reported should strive to be minimal.

The second problem is that the algorithm must first **seed** itself by identifying lines which occur exactly once in each file. It is possible for it to fail to match **any** lines even when there is an obvious match.

Consider the two files with the following contents:

aaa	bbb
xxx	xxx
xxx	xxx
yyy	zzz

The algorithm will fail to match the blocks of **xxx** lines since it does not find any lines which occur exactly once in each file to seed its block matching algorithm. Such an arrangement of lines in normal text files is highly improbable. However, if something similar to this occurs, you will be aware of the reason for it.

B.2 DIFF Memory Size Considerations

In addition to these algorithmic points, there are some considerations of memory size which you may wish to keep in mind. The program uses two tables (one for each file) to keep track of the lines read in from each file. The

default size of each of these tables is 1000, however, as mentioned above this may be varied by use of the **-l** option.

When the **-w** option is invoked, additional memory is required since the program needs to keep a representative of each line in both its compressed and original form. Thus the **-w** option forces the creation of a third table used in the storage and comparison of lines.

Note, in all cases only the minimal number of strings will be stored since only one copy of a line's text is ever stored. The program also will dynamically allocate storage for any line text it needs to store.

Finally, the program uses an input/output buffer of 8K in size. There are, therefore, several circumstances in which the **DIFF** utility may run out of space. Should **DIFF** run out of memory space, there are several responses which you may employ. These are:

- You may attempt to decrease the size of the line tables by employing the **-l** option.
- You may try to use **DIFF** without using the **-w** option.
- You may decrease the size of the I/O buffer by using the **-b** option.

Note that decreasing the size of the I/O buffer may result in some speed degradation since more disk accesses may be required. However, if you are using a hard disk, this factor can be considered negligible.

Index

A

B

C

D

E

F

G

H

I

L

Commands

Commands

Lattice C Commands

Command Summary for the Lattice Amiga C Compiler

Lattice, Inc.
2500 S. Highland Avenue
Lombard, IL 60148
USA

A Subsidiary of SAS Institute Inc.

Lattice C Commands Manual

This manual was formatted using **HighStyle**® by Lattice, Inc.

SAS/C® Commands Manual

Command Summary for the SAS/C® Compiler for AmigaDOS™

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

SAS/C® Commands Manual

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Table of Contents

Section 1

Introduction to Commands

This section summarizes all of the commands in the Lattice Amiga C Compiler package. Appendix A summarizes the options for the compiler commands, **lc**, **lc1**, and **lc2**, in a single table.

The following is an index to all of the commands in the Lattice Amiga C Compiler package:

COMMAND	DESCRIPTION	PAGE
asm	Lattice 68000 Macro Assembler	C4
blink	Linker for the Lattice AmigaDOS compiler	C9
build	Inserts lines of text into a given file	C22
cpr	CodePRobe source-level debugger	C23
cxref	Generates a cross-reference listing for C language source files	C27
diff	Determines the differences between two files	C29
extract	Prints the names of files in specified directory	C31
fd2pragma	Pragma Generator	C32
files	Search, copy, or erase files or directories	C33
grep	Global regular expression search and print	C35
lc	Lattice C Compiler	C40
lc1	Compiler pass 1	C61
lc1b	Big comiler pass 1	C61
lc2	Compiler code generator	C63
lcompact	Header file compressor	C64
lmk	Maintain and update records of file dependencies	C66
lprof	Execution Profiler	C73
lse	Edit user files	C75
lstat	Profile reader and printer	C76
omd	Object Module Disassembler	C80
oml	Object Module Librarian	C82
splat	Stream editor for performing character substitution	C89
tb	Traceback Utility	C90
touch	Adjust the time stamp on specified files to the system time	C93
wc	Tabulate the number of characters, words, lines within file	C94

Section 2

Lattice Amiga Compiler Commands

NAME

asm Lattice 68000 Macro Assembler

SYNOPSIS

```
asm [>listfile] [options] filename
```

DESCRIPTION

The AmigaDOS macro assembler is designed to generate assembly language components for inclusion in C programs, but it can also be used to generate programs entirely written in assembly language. It supports the full set of 68000, 68020, 68030, and 68881 mnemonics, as well as an extensive set of assembler directives and a powerful macro facility.

The assembler reads an assembly language source file and produces an object file in the Amiga object file format, along with an optional listing of the source and assembled code. The format of the command to invoke the assembler is:

```
asm [>listfile] [options] filename
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

>listfile Causes the listing and error message output of the assembler to be directed to the specified file.

options Assembler options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-c Specifies that the sections designated by the characters which immediately follow are to be loaded into

memory addressable by the Amiga's custom hardware. This is necessary for screen image and audio data. The **-c** option must be immediately followed by one or more of the following letters in any order:

b bss or uninitialized data

c code segment

d data segment

The default action is to load the segments into memory not addressable by the custom hardware if it is available. This is generally desirable because it avoids bus contention between the processor and the custom hardware. Image and audio data must be loaded into memory which accessible by the custom hardware. For example:

```
-ccdb
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

-d This option has two uses. It activates the debugging mode and it defines symbols.

The -d option can also be used to define symbols in the following ways.

-dsymbol Causes **symbol** to be defined as if your source file had the statement:

```
symbol EQU 1
```

-dsymbol=value — Causes **symbol** to be defined as if your source file had the statement

```
symbol EQU value
```

-h — Specifies that the sections designated by the characters which immediately follow are to be loaded into memory not addressable by the Amiga's custom hardware. The **-h** option must be immediately followed by one or more of the following letters in any order:

b	bss or uninitialized data
c	code segment
d	data segment

The default action is to load the segments into memory not addressable by the custom hardware if possible. This option prevents the specified segments from being loaded into chip addressable memory even if no other memory is available. This situation can cause programs to not execute if external high-speed memory is not available on the machine. As an example:

```
-hcdb
```

would cause all three sections to be loaded only in high-speed ram.

-iprefix — Specifies that INCLUDE files are to be searched for by prefixing the filename with the string **prefix**, unless the filename in the INCLUDE statement is already prefixed by a drive or directory specifier. Up

to 4 different **-i** strings may be specified in the same command. When an unprefixed INCLUDE filename is encountered, the current directory is searched first; then file names are constructed and searched for, using prefixes specified in **-i** options, in the same left-to-right order as they were supplied on the command line. No intervening blanks are permitted in the string following the **-i**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

-l[list] Causes a listing of the source file to be written to the standard output. The listing displays the appropriate location counter and data generation information alongside the assembly source. One or more of the following characters may be appended to the **-l** option, with the following effects:

x Lists the expansion text for macros.

m Lists additional data generated for source lines which cannot be accommodated alongside the original source line (i.e. allows multiple listing lines for each source line).

i Lists the source for text from INCLUDE files as well as the original source file.

-m0 Used for 68000 target. Provides warning flags if you attempt to use 68020 only instructions. This is the default case.

-m2 Used for 68020 target. Turns off the warnings supplied in the -m0 option.

-oprefix Specifies that the output filename (the .o file) is to be formed by prefixing the input filename (the .a file which is being assembled) with **prefix**. Any drive or directory prefixes originally attached to the input

filename are discarded before the new prefix is added. No intervening blanks are permitted in the string following the **-o**. Note that if a directory name is to be specified as a prefix, a trailing slash must be supplied.

-s Includes the section name at the beginning of each hunk.

-u This option is needed to assemble the startup code. It prefixes external references with an underscore (_). If all external references to C files have been prefixed with an underscore, the option is not needed.

-w This option works like the -d option on *shortint* (e.g., **-d shortint**).

filename Specifies the name of the assembly language source file which is to be assembled; this is the only command line field which must be present. If the filename is specified without an extension, .a is assumed. The object file created by the assembler will have the same name as the source file, except that .o will be supplied in place of any extension.

EXAMPLE

The following command causes the assembly language source file **modn.a** to be assembled, producing the object file **modn.o**. A listing of the source file, along with any error messages generated, will be written to the file **modn.lst**.

```
asm >modn.lst -l modn
```

NAME

blink Linker for the Lattice AmigaDOS compiler

SYNOPSIS

```
BLINK [FROM][ROOT] beta [TO alpha][WITH alpha]
[VER alpha] [LIBRARY | LIB beta][MAP alpha map_options]
[XREF alpha][options]
```

DESCRIPTION

The **blink** command is for the Lattice linker. The options for the command are:

alpha means a single file.

beta means one or more file names separated by a comma, plus sign, or space.

BLINK provides a large number of keywords to give the programmer a wide number of options. Although some of these may seem superfluous, the intention is to provide the programmer with as many options as possible, even if some of these options appear rather obscure. The following keywords are recognized by BLINK:

ADDSYM This causes BLINK to emit HUNK_SYMBOL records for all symbols in the input object files regardless of whether or not the input object file was compiled with the *-d* option.

This is extremely useful for use with a symbolic debugger such as Lattice CodePRobe.

BATCH This causes BLINK to supply the default value of 0 for all undefined symbols. Normally, BLINK will pause after each undefined symbol to give you an opportunity to correct the error. If you specify the BATCH option, it will not pause.

BUFSIZE *n*	Sets the I/O buffer size for BLINK. By default, all I/O is done in 488 character blocks. However, you can alter the I/O performance by adjusting this number. In general the default is sufficient, but may be decreased if memory is severely constrained (at the price of performance). Note that one buffer is allocated for each file BLINK has open at any one time (a maximum of 4).
CHIP	Specifies that all hunks are to be placed in Chip or display memory regardless of the input object hunk specifications.
DEFINE *symbol=symbol*	This defines a symbol to be used in the linking process. This is particularly useful in conjunction with the PRELINK option to force certain routines to be pulled from the library even though no references to them exists.
DEFINE *symbol=val*	This assigns a symbol to a value to be used in the linking process.
FANCY	Enables usage of printer control characters in the map file.
FAST	Specifies that all hunks are to be placed in Fast or expansion memory regardless of the input object hunk specifications.
FASTER	A do-nothing option that is included only for ALINK compatibility.
FROM *files*	Specifies the object files that are the primary input to the linker. These object files will always be copied to the root of the object module. You must specify at least one object file for the root. If it appears as the first option to BLINK then the FROM keyword is optional and may be omitted. ROOT is an acceptable synonym for FROM. FROM may be used more than once with the files for each FROM adding to the list of files to be linked.

FWIDTH *n*	Total number of columns in the map file.
HEIGHT *n*	Total number of lines on a page in a map file. Refer to the section on map files below.
HWIDTH *n*	Hunk name field width, in columns, in the map file.
INDENT *n*	Number of columns to indent on a line in the map file.
LIB *files*	Specifies the files to be scanned as libraries. Only referenced segments from library files will be included in the final object module. LIBRARY is a valid synonym for LIB.
LIBRARY *files*	See LIB.
MAP *file options*	Specifies a file to which a map is to be written. Options controls which parts of the map will be written. See the MAP section for more information.
MAXHUNK *n*	Limits the maximum size hunk that Blink will create when coalescing hunks. This can be used to control fragmentation of memory. The default size is no limit on hunk size.
ND	See NODEBUG.
NOALVS	Prevents BLINK from creating ALVs to resolve 16 bit PC relative code. This can be used to stop BLINK from creating a non-relocatable object from what was intended to be relocatable code.
NODEBUG	Suppresses any HUNK_DEBUG, symbol table information or hunk names in the final object file. This is equivalent to the object file that would be produced if STRIPA were run on the final object file. ND is a valid synonym for NODEBUG.
OVERLAY	Specifies the start of an overlay tree terminated by a line consisting of a single hash sign '#'. See the OVERLAY section for more information.

OVLYMGR *file*	Directs BLINK to use <file> as the Overlay Manager, instead of the default. The file should consist of a single code hunk of name *NTRYHUNK*.
PRELINK	Causes BLINK to output an object module with references and definitions still intact so that it can be linked later on to produce a final executable file. This is designed for development of a large project where the programmer is only changing a single source module. This is described in more detail below. Note that this doesn't function with overlays.
PLAIN	Turns off the FANCY map file option.
PWIDTH *n*	Width of program unit name fields in the map file.
Quiet	Causes no messages to come out except in the case of an error.
ROOT *files*	Specifies the object files that are the primary input to the linker. These object files will always be copied to the root of the object module. You must specify at least one object file for the root. If it appears as the first option to BLINK then the ROOT keyword is optional and may be omitted. FROM is an acceptable synonym for ROOT. ROOT may be used more than once with the files for each ROOT adding to the list of files to be linked.
SC	See SMALLCODE.
SD	See SMALLDATA.
SWIDTH *n*	Width of symbol names field in the map file.
SMALLCODE	Causes all CODE hunks to be coalesced into a single hunk. SC is a valid synonym for SMALLCODE.
SMALLDATA	Causes all DATA and BSS sections to be coalesced into a single hunk. This is useful for combining all data

	hunks from a program into a single hunk, decreasing load time but potentially producing larger hunks that are difficult to scatter load. SD is a valid synonym for SMALLDATA.
TO *file*	Specifies target object module to create. If omitted it defaults to the same name as the first object module listed on a FROM option with its .o suffix removed.
VER *file*	See VERIFY.
VERBOSE	Causes BLINK to print out the names of each file as it processes it.
VERIFY *file*	A destination file to contain all linker output messages. If you do not specify it then all messages go to the terminal. VER is a valid synonym for VERIFY.
WIDTH *n*	Sets the maximum line length for the map and cross reference listings. This is useful when sending the output to a device which has different line length requirements. If not specified it defaults to 80.
WITH *file*	Specifies a file containing BLINK command options to be processed for this link. More than one WITH file may be specified as may WITH files contain WITH statements. The contents of all with files will be treated as if they were specified on the BLINK command line.
XREF *file*	Specifies a file to which the cross reference information will be written. If not specified and a cross reference is requested with the MAP option, the cross reference listing will appear as part of the map file.

Compiler Options and Overlays

The -b Option	This option causes all data within a program, whether in a root node or an overlay node (if any), to be merged into one large block which is referenced via 16-bit references. Overlays in BLINK have NO effect on this behavior.

The -r Option

The -r option causes all subroutine references to be via 16-bit relative branches. This modification is extended to overlay nodes, but not across them. Routines in one overlay node being referenced by routines in other overlay nodes are referenced via an *Overlay Call Vector* or *OCV*, which is automatically created for you. An *OCV* is responsible for telling the Overlay Manager to load the desired overlay node.

References within a node which span more than 32k are given an *ALV*, or *Automatic Link Vector*, again, automatically created for you, and guaranteed to transfer control to the actual routine you were trying to reach.

> NOTE: If you use the -r compiler option and overlays, you will find the *OCV*'s being referenced by *ALV*'s when the -r compiler option is used.

Other BLINK Options and Overlays

SMALLCODE

When the SMALLCODE option is used with overlays, BLINK tries to merge as much of the code as possible, without crossing overlay node boundaries. No code will ever move out of the node it was defined to be in.

SMALLDATA

When the SMALLDATA option is used with overlays, BLINK merges data hunks together within each overlay node and the root. Data does NOT get moved across node boundaries.

There is an exception – if the -b compiler option was in effect on any or all object files, the data segments of those object files will ALL be moved into the root node.

Map Files

The command line structure for these files takes the form:

MAP <map file> <map options>

These files provide a large number of options for the programmer to customize the output format. The *<map options>* are:

F H L O S X FANCY PLAIN

Note the use of just one character on most options. The options may be used individually or combined in the general form:

MAP [[filename],option,option,...]

where:

filename	map output file
option	letter of report to produce (see below)

The single character options are shown below:

F	MapFile
H	MapHunk
L	MapLibraries
O	MapOvly
S	MapSym
X	MapXref

PLAIN and *FANCY* are used to select the sub-options below:

FANCY	flag to allow printer control characters in the map file, *(default condition)*
FWIDTH *n*	width of file names *(default 16)*
HEIGHT *n*	lines on a page in map file, 0 indicates no pagination *(default 55)*
HWIDTH *n*	width of hunk names *(default 8)*
INDENT *n*	columns to indent on a line, this is included in width *(default 0)*

PLAIN	turns off the FANCY option
PWIDTH *n*	width of program unit names *(default 8)*
SWIDTH *n*	width of symbol names *(default 8)*
WIDTH *n*	columns allowed in map file *(default 80)*

Special Names

The following hunk section names are reserved to have special meanings:

NTRYHUNK — There can only be one of these and the hunk with this name will be the FIRST hunk in the output executable. Usually this is used only by the overlay manager's code hunk.

__MERGED — Only data and BSS Hunks with this name are permitted. If a data or BSS hunk has this name it WILL be merged with any other hunks with the same name, and moved into the root node (if overlay nodes exist).

_NOMERGE — Any hunk with this name will NEVER be merged with any other hunk. It will occupy a single, separate hunk in the final output executable.

The name *BLINKWITH:* is a special ASSIGN name to BLINK – if defined, it is assumed to be a default WITH file. This file appears similar to any other WITH file, and its contents are always applied. However, it does NOT prevent the usage of the command line or another WITH file. Command lines and non-default WITH files do NOT override the default WITH file – they are effectively appended to it.

Thus, creating a default OVERLAY tree is not a good idea and is flagged as an error. The OVERLAY directive should only be used in a non-default WITH file.

EXAMPLE

Example WITH files

```
from c.o
vt100.o
init.o window.o
xmodem.o remote.o kermit.o script.o
lib lib:lc.lib lib:amiga.lib
VERBOSE
SMALLCODE
SMALLDATA
to vt100
```

An alternative would be a header WITH file which you can append the specifics to the end of the file and save some typing each time you needed to create a WITH file.

```
LIBRARY
ram:lc.lib
ram:amiga.lib
SMALLDATA
SMALLCODE
MAP
ram:mymap
VERBOSE
XREF ram:myxref
FROM c.o
vt100.0 init.o window.o xmodem.o remote.o kermit.o script.o
```

In the above example you will notice that no TO file was specified. BLINK will automatically create a TO file for you in the current directory with the name of the first module in your FROM list not including the startup routine in this case c.o. In the above example the executable will be called VT100 without the .o extension.

Overlay Examples

Example 1 – "Hello, World!"

This example is based on the following functions:

```
#include <stdio.h>

void main()
  {
    Hello();
    World();
  }

void SayWord(s)
   char *s;
   {
    printf("%s", s);
   }

void Hello()
   {
    SayWord("\nHello, ");
   }

void World()
   {
    SayWord("World!\n");
   }
```

where the first two functions are defined in the file HW.C, and the other two are defined in HELLO.C and WORLD.C, respectively.

While decidedly extravagant for a "Hello, World!" program, it can be used to demonstrate the basic operation of Overlays with BLINK.

After compilation, the object files HW.O, HELLO.O, and WORLD.O are available for linkage. The following WITH file is given to BLINK to build the file:

```
ROOT /lib/c.o hw.o
OVERLAY
hello.o
world.o
```

```
#
LIBRARY  /lib/lc.lib  /lib/amiga.lib
MAP hw.map hxsflo
TO hw
```

which corresponds to the following tree:

```
c.o,hw.o,lc.lib,amiga.lib
     |          |
  hello.o    world.o
```

BLINK evaluates the WITH file, creating an executable called *HW*, and a map file with all possible map information, called *HW.MAP*. The executable consists of a *root* node, containing all of the code and data in *C.O* and *HW.O*, and two separate *overlay* nodes, containing the code and data of *HELLO.O* and *WORLD.O*, respectively. The data and code for the libraries *LC.LIB* and *AMIGA.LIB* are also in the *root* node. To run the program, just do what you normally do to run a program – the overlay operation is automatic, and you will see the message, "Hello, World!" printed out on your display.

Example 2 – A General Editor

Sorry, but there is no source code available for this one. Suppose you had written a new program to edit both graphics and text in a window. Immediately, two overlay nodes should come to mind:

- One node for the graphics-specific editing routines.
- Another for the text-specific routines.

The reason is because if you are busy editing text, you do not need to keep the graphics routines loaded and vice-versa, giving you more memory for your data structures.

Another feature of the program would be to print the edited data. For this example, the internal data format is so elegant that you only need one print procedure to print both graphics and text (the text is treated as graphics). Unfortunately, the print routine occupies 60 kbytes of memory. Fortunately, it's only printing – it's only done when requested. Another overlay node is born, probably out of the *root* node. The tree for this application resembles the following:

```
root, with miscellaneous I/O routines and main entry point
            |                      |              |
         graphics                 text         printing
```

meaning a WITH file as follows:

```
ROOT /lib/c.o omniedit.o
OVERLAY
graphedit.o
testedit.o
print.o
#
LIBRARY /lib/lc.lib /lib/amiga.lib
TO omniedit
```

Consider a situation where the graphics editor uses various iconic tools, such as a paint brush, a fine-detail tool, and a flood-fill tool. Only one tool can be used at a time, so why keep the others in memory if spare memory is the prime goal? No good reason, so the routines which support these tools can be overlays hanging off the graphics overlay node, and the graphics node itself can just contain the code for refreshing the display window. This make the tree more like the following:

```
root, with miscellaneous I/O routines and main entry point
       |                        |                    |
   graphics                    text               printing
for window
refreshing
       |
paintbrush   fine-detail   flood-fill
```

which yields:

```
ROOT /lib/c.o omniedit.o
OVERLAY
graphedit.o
*paintbrush.o
*finedetail.o
*floodfill.o
testedit.o
```

```
print.o
#
LIBRARY /lib/lc.lib /lib/amiga.lib
TO omniedit
```

This refinement process could continue for several more levels. The point is that it would easily adapt to overlays, which would minimize the memory requirements of the code of the program, and leave more for the application's data, and for the code and data of other applications running concurrently.

Some example default and non-default WITH files look like this:

Example 3.

```
FROM /lib/c.o
LIB /lib/amiga.lib /lib/lc.lib
SMALLCODE  SMALLDATA  BATCH
```

Example 4.

```
FROM maina.o mainb.o
OVERLAY
ov1a.o ov1b.o
*ov2a.o
ov1c.o
#
MAP miffle.map hsflxo
TO miffle
```

Note that the OVLYMGR directive can be used in the default WITH file, although it is not recommended (like OVERLAY).

NAME

build Inserts lines of text into a given file

SYNOPSIS

```
build >alpha beta
```

DESCRIPTION

This utility takes input from the standard input device - the keyboard. The contents of the input stream are then inserted between the current line in the file **beta**. After receipt of the input terminator characters (Ctrl \), **build** moves down to the next line in the file named **beta**. The process is repeated until the end-of-file position is reached. The resulting file **alpha** then contains the original lines of characters from the file named **beta** interleaved with lines of input from the keyboard.

Blink comes with an added feature for generating resident libraries. Here's how to do it:

- All modules in the library that reference global data must be compiled with the "-ml" option.
- All functions that are callable from outside the library must be declared with the "__saveds" keyword.
- Create a .fd file for your library.
- Link with the following command line:

```
blink LIBPREFIX _LIB LIBFD fdfile TO libname FROM libent.o
libinit.o libmodules
```

where

LIBPREFIX	is an optional keyword that specifies what prefix to add to the functions listed in the .fd file. *_LIB* is the prefix in this example. The default is simply an _. Note that without this keyword, you can't use stubs to call this library. Also remember that the compiler adds an underscore to everything, so if you define your function as **LIBtest()** in your C code, the prefix should be **_LIB**.
LIBFD	is a new keyword used to specify the name of the .fd file (fdfile).
TO	is used to specify the resulting library filename (libname).
libent.o	contains the RomTag for the library, and MUST BE THE FIRST MODULE (ala c.o).
libinit.o	contains Open, Close, Expunge, and other initialization routines, and THIS MUST BE THE SECOND MODULE.
libmodules	are user modules.

Class: Lattice

NAME

build Inserts lines of text into a given file

SYNOPSIS

```
build >alpha beta
```

DESCRIPTION

This utility takes input from the standard input device - the keyboard. The contents of the input stream are then inserted between the current line in the file **beta**. After receipt of the input terminator characters (Ctrl \) **build** moves down to the next line in the file named **beta**. The process is repeated until the end-of-file position is reached. The resulting file **alpha** then contains the original lines of characters from the file named **beta** interleaved with lines of input from the keyboard.

NAME

cpr CodePRobe source-level debugger

SYNOPSIS

```
cpr [options] application name [application arguments]
```

DESCRIPTION

CodePRobe is a source-level debugger that allows you to examine the behavior of a program written in the C language using the constructs of the language. **CodePRobe** allows you to run a program in a controlled environment where execution can be stopped at any point. For instance, breakpoints can be set at particular source lines or within modules of interest.

CodePRobe comprises four windows: the Dialog Window and the Source Window open by default, while the Watch Window and the Register Window may be opened by function keys.

An application program can be run under debugger control by typing "cpr" followed by the command line normally used to invoke the program. To display the C source symbols and attributes of your program, you must take certain steps when you compile and link the program. (Refer to the descriptions of debugger options in the discussion of the **lc** command in this section.)

When the **cpr** command is invoked, **CodePRobe** makes the following assumptions:

- The program specified on the command line exists in the current directory or in the execution path defined by AmigaDOS.
- The modules making up the program to be debugged have been compiled with one or more of the debugging options and have been linked using the ADDSYM option of the linker.
- Any source files you want to access (e.g., to set breakpoints at source lines) exist in either the current directory, your special source path list, or the directory in which the module was compiled.

To stop **CodePRobe** at any time, you need only issue the **quit** command (abbreviated as **q**). You can also restart your program without quitting **CodePRobe** by means of the **restart** command.

The **CodePRobe** environment can be customized by a number of available command-line options (If there are any command-line options for the program you are debugging, you must include these on the command line as well.) You can use the following command-line options when invoking **CodePRobe**:

Window Dimension options — Use the -wdialog, -wregister, -wsource, and -wwatch options to specify start-up window coordinates for the Dialog, Register, Source and Watch Windows, respectively. Window coordinates are measured in character positions. A width or height of 0 causes the window to extend to the screen border on the right or bottom, respectively. The option names can be abbreviated to two letters:

> **-wd**[ialog] *left top width height*
> **-wr**[egister] *left top width height*
> **-ws**[ource] *left top width height*
> **-ww**[atch] *left top width height*

For example:

```
cpr -ws 0 1 50 12 -wd 0 13 50 0 ftoc
```

Workbench option — Use the **-w** option to setup **CodePRobe** in the Workbench screen instead of a new screen.

Interlace option — Use the **-i** option to setup a screen in interlace mode. By default, **CodePRobe** opens a new screen using the specifications set up by Preferences for Workbench screens. If you wish to set up a screen in interlace mode, include the **-i** option before typing the application command name.

Command option — Use the -command option (followed by specific commands) to execute debugger commands on startup. The

commands are executed after the "go main" if the **-startup** option is not specified, or after the profile script if it is specified. (Profile scripts are discussed later in the description.) For example:

```
cpr -command "proceed; display fahr"
```

would execute to main, step over 1 line of code and display the variable *fahr* in the Dialog Window before giving control to the user.

Line option

Use the **-line** option to start **CodePRobe** in line mode. Line mode causes the debugger to run in the current CLI window. Only a Dialog Window is provided. Line mode is probably only useful for running from a script file, redirecting output using the CLI redirection facility, or running from a remote terminal over a communications port.

Startup option

The **-startup** option suppresses the automatic "go main." that is normally executed by the debugger on startup. This option is useful if your application redefines *main* or *c.a* so that no main function exists or if you wish to step through such code. If this option is used, no initialization of any kind will have been performed by the application process before control returns to the user.

If the **quit**, **start**, or **restart** commands are invoked and an application process has not exited, the debugger normally calls *exit* to cleanup any process resources that may not have been freed. However, if the **-startup** option was used, *exit* will not be called.

Temporary file option

In order to access debugging information in an optimal manner, **CodePRobe** creates a temporary file called a "spill file" to store information for quick access. Normally this file is located in **ram:**. The **-temp** *filename* option opens a temporary spill file called *filename* instead of the

default *ram:cpr.tmp*. This option may be useful to move the spill file to some other location if your system has a limited amount of memory.

Since you will probably want **CodePRobe** to start the debugger with the same options most of the time, **CodePRobe** allows you to create debugger command script files to be executed each time the debugger is invoked. The profile scripts are executed after the initial "go main," but before any commands specified with the **-command** option are executed.

Profile scripts must be named *profile.cpr*. **CodePRobe** first looks for a script in the **s:** directory and, if successful, executes it; then **CodePRobe** searches through the execution path defined by AmigaDOS. It will execute only the first script encountered in the path. (Note that the path always includes the current directory first and c: last).

SEE

lc, blink

NAME

cxref — Generates a cross-reference listing for C language source files

SYNOPSIS

```
cxref >destn source [options]
```

DESCRIPTION

This utility will take the names of files from a specified directory and/or drive unit path and generate a file. The output can be redirected using the AmigaDOS redirection command to another file or device. Where no directory is specified, the current directory is used. The generated file will contain both a program listing as well as several cross-reference tables consisting of:

- Pre-processor defined identifiers.
- Functions.
- Labels.
- Structure identifiers.
- Identifiers.

With the exception of the **-i** option, any command line option or flag can be entered discretely or in conjunction. For instance the following are equivalent and permissible commands:

```
cxref >destination source -pn

cxref >destination source -p -n
```

Flags may be specified in any order provided a flag is placed adjacent to its parameters, if any. The following summarizes all of the available command line options for **cxref**.

-ipath Specifies the #include files to be processed for listing and cross-reference. The specified paths and **include** directives

are taken if this option is followed by the appropriate command, e.g. **idf0:** or **idir1** etc.

-lnn	Defines the length of a printed page, the default being 66 lines per page. The minimum value is 10 lines per page. The syntax used is **-lnn**, where **nn** represents the numeric value.
-n	Removes all page headings - the default is to place a heading on each page.
-o	Omits any identifiers, both normal and structure.
-p	Suppresses generatation of a cross-reference report.
-r	Includes any C language reserved words into the cross-reference. Selecting this option will ensure that pre-processor commands and reserved keywords are additionally listed.
-wnn	Defines the width of a printed page, the default being 80 characters per line. The minimum value is 60 characters per line, the maximum is 132 characters per line. The syntax used is **-wnn**, where **nn** represents the numeric value.
-x	Generates a cross-reference report with the program listing.

NAME

diff Determines the differences between two files

SYNOPSIS

```
diff [-b nn] [-c] [-F nn] [-L nn] [-l nn]
     [-o file] [-p] [-q] [-w] file1 file2 ...
```

DESCRIPTION

This utility takes the two specified files and reports any textual differences between them. The default output is the screen, although the AmigaDOS redirection command is supported.

The output file consists of three types of change blocks - append, change or delete. These blocks contain the information relating to any differences between the two files.

diff does provide for error message generation. Full details of these is given in the main section relating to **diff**.

Command line options, or flags, can be entered discretely or in conjunction:

```
diff -pn file1 file2
```

and

```
diff -p -n file1 file2
```

are both allowed. Flags may be specified in any order provided a flag is placed adjacent to its parameters, if any. The following summarizes all of the available command line options for **diff**.

-b**nn** This option determines the size of the I/O buffer, where **nn** represents the number of bytes. The default I/O buffer size is 4K.

-c	This option will display only those lines common to both files.
-Fnn	This option will determine the column position of the **first** character for file comparison purposes. For example, setting the **nn** value to 25 will cause **diff** to ignore the first 25 characters on each line.
-Lnn	This option will determine the column position of the **last** character for file comparison purposes. For example, setting the **nn** value to 75 will cause **diff** to ignore characters beyond this column position on each line.
-lnn	This option will define the number of lines per file that can be handled by **diff**. The default is 2000 lines - the maximum number of lines is limited by memory availability.
-o file	This option will direct the output to the specified file or device - this is an alternative to the AmigaDOS redirection command >.
-p	This option will remove any unprintable characters from the input stream. It is identical to the **-p** option of **GREP**.
-q	This option will not output any messages when there are no differences between the specified files.
-w	This option will ignore differences related solely to space or tab characters and reduce sequences of these characters to one character.

NAME

extract Prints the names of files in specified directory

SYNOPSIS

```
extract [-b | -r] [-n] pattern1 pattern2
```

DESCRIPTION

This utility will take the names of files in a specified directory and list them to the standard output device - the screen. The output can be redirected using the AmigaDOS redirection command to another file or device. Where no directory is specified, the current directory is used. The use of *wildcard* characters is supported by this utility.

Command line options, or flags, can be entered discretely or in conjunction:

```
extract -bn pattern1 pattern2
```

and

```
extract -b -n pattern1 pattern2
```

are both allowed. Flags may be specified in any order provided a flag is placed adjacent to its parameters, if any. The following summarizes all of the available command line options for **extract**.

-b This option will not display any filename extension.

-n This option will not sort filenames into alphabetical order before output.

-r This option will not display filename extensions and prefixes.

NAME

fd2pragma Pragma Generator

SYNOPSIS

```
fd2pragma infile outfile
```

DESCRIPTION

This command is used to create a file containing **#pragma** commands to allow the compiler to generate inline subroutine calls to an external library.

The **infile** gives the complete name of the input **.fd** (*Function Description*) file. This file is in the same format as supplied by Commodore on the Extras disk for use with AmigaBasic.

The **fd2pragma** program will convert the function descriptions in the source **.fd** file into the appropriate *#pragma* statements for the compiler to generate inline calls. Note that any subroutines requiring more than 4 parameters will not have *#pragma* statements generated for them because of the limitation of 4 parameters in the compiler.

Any errors encountered in the **infile** will be reported to the console.

SEE

lc, lc1

NAME

files Search, copy, or erase files or directories

SYNOPSIS

```
files [-b] [-copy dest] [-days nn] [-erase] [-n]
      [-name file/dir] [-m] [-newer file]
      [-older file] [-pat pattern] [-r] [-rerase]
      [-size nn] [-type arg] [-v] files dirs
```

DESCRIPTION

This utility permits a full range of file or directory manipulations. These manipulations can be either searching, copying or deletion of files and directories. In addition, **files** supports a *recursive erase* feature which allows the deletion of an entire directory structure on a single command. **files** also supports the AmigaDOS redirection command.

The following summarizes all of the available command line options for **files**.

-b	This option will only output the basename of files - in other words, filename extensions are ignored.
-copy destination	This option will copy files or directories and associated structures to the specified destination.
-days nn	This option will display files or directories modified within the last **nn** days.
-erase	This option will erase the specified files or directories and associated structures.
-n	This option will not descend recursively into a directory structure to copy, erase or search.
-name file/dir	This option will search for a specified filename or directory. AmigaDOS wildcards are supported by this option. This option cannot be used in conjunction with the **-pat** option.

-m	This option will use the DOS wildcard conventions of asterisks, rather than the AmigaDOS conventions of hash symbol (#) and question mark (?).
-newer file	This option will search for the specified files which have a date stamp later than a specified file.
-older file	This option will search for the specified files which have a date stamp earlier than a specified file.
-pat pattern	This option will search for a specified file using a regular expression pattern based on the conventions followed by **GREP**. This option cannot be used in conjunction with the **-name** option.
-r	This option will search on the root name of the file - in other words, ignore any filename extensions.
-rerase	This option will erase the selected files and directories. This includes a recursive descent into any subdirectories to delete files and directories.
-size bytes	This option will search for a file using the specified minimum size of the file in bytes as the criterion.
-type argument	This option will search for either files or directories - a **d** argument means a directory, an **f** argument means a file.
-v	This option will echo the names of any file or directory being deleted to the screen. By default **files** does not display these names to screen.

NAME

grep Global regular expression search and print

SYNOPSIS

```
grep [>outfile] [-c] [-f] [-n] [-p] [-q] [-s]
     [-v] [-V] [-$] pattern file1 file2 ...
```

DESCRIPTION

Command line options, or flags, can be entered discretely or in conjunction:

```
grep >alpha -pn beta
```

and

```
grep >alpha -p -n beta
```

are both allowed. Flags may be specified in any order provided a flag is placed adjacent to its parameters, if any. The following summarizes all of the available command line options for **grep**.

-c This option will print a count of the matched lines.

-f This option will print the names of the files in which a match of the pattern was found.

-n This option will turn off the numbering of matched lines - the default and opposite of the UNIX convention, in that each matched line is preceded by its line number when displayed.

-p This option will filter out non-printable characters. This option is useful when operating **grep** on AmigaDOS binary files in which control characters may occur. Only the printable ASCII characters will be displayed.

-q This option will not display filenames or line numbers.

-s This option will show all filenames - normally **grep** will display only those filenames where it has found a match for the expression. This option forces the printing of filenames in which no match was found.

-v This option will print only the lines in which a match of the pattern is **not** found. This option can also be used with **-n**, **-f** and **-c**.

-V This option will print the version number of **grep**.

-$ This option will ignore character case distinctions. Normally **grep** distinguishes between upper and lower-case. Use of this flag forces **grep** to ignore the distinction. For example, each of **int**, **INT**, **Int**, **INt**, etc. would match the pattern **int** if the **-$** option were used.

This version of **grep** is very much like the UNIX grep, but with a few exceptions. A slightly less general class of patterns (regular expressions) is supported. In particular, the UNIX patterns of the form:

```
\{m\}   or   \{m,\}   or   \{m,n\}
```

which match ranges of occurrences of patterns are **not** recognized by this **grep**. This also applies to patterns of the form:

```
\( ... \)      or      \n
```

where **n** is a numeral, and **\n** is intended to match the same string of characters as was matched by the expression enclosed between \(and \).

Aside from these departures, this **grep** has all the features of the UNIX grep. In particular:

1. An ordinary character is a one-character pattern which matches itself.

2. A backslash (\) followed by any special character is a one-character pattern which matches the special character itself, **except** when it is used in an **escape** sequence for a non-graphic character as described below. The special characters are:

\ . * [,

Unlike UNIX, the characters:

. * \

remain special even within square brackets, although [does not.

^ which is special at the **beginning** of an **entire** pattern.

$ which is special at the **end** of an **entire** pattern (see below).

! which is special immediately following a left bracket [(see below).

3. A period (.) is a one-character pattern that matches any character except newline or end-of-string.

4. A non-empty string of characters enclosed in square brackets [ ] is a one-character pattern which matches **any one** character in that string. However, if the first character of the string is **!** then the one-character pattern matches any character **except** newline or end-of-string and the remaining characters in the string. Thus **!** is a *not* sign applied to a character class. Note that this is a notational departure from UNIX, which uses ^ for this application (thus rendering ^ ambiguous).

5. Also supported are such character classes as:

   [a-z]

   indicating all of the letters *a* to *z*. The only restriction in such cases is that the first character must occur alphabetically prior to the second.

6. **grep** recognizes certain non-graphic characters denoted in the manner of *Kernighan & Ritchie*. In particular, the expression on the left below may be used to denote a pattern which is matched by the character described on the right:

\n	newline
\t	tab
\b	backspace

\r	carriage return
****	backslash
\xmn	character whose hexadecimal number is denoted by the hexadecimal numerals **m** and **n** (in sequence).

7. Patterns containing white space (spaces and tabs) may be specified to **grep** by enclosing them in double quotation marks (" "). Thus a string may be **grep**ped by:

```
grep "this is a string" foo.c
```

The manner in which the command line given to AmigaDOS determines the behavior of **grep**. Thus, any tab in the command line will remain a tab when parsed by **grep** - it will not be converted to spaces. Spaces, in addition to any combination of tabs and spaces, will be passed unchanged to **grep** from the command line. It is also acceptable to use the escaped form of the tab character **\t**, inside a quoted string passed to **grep**. If it is necessary for **grep** to parse an expression containing the double quote itself, this character should be escaped with the backslash as shown in the following example:

```
grep "this : \" is a double quote" foo.c
```

8. If **r** is a pattern, then:

```
r*
```

matches **zero or more** occurrences of **r**. If there is a choice, then the longest leftmost string that matches **r** is chosen.

9. If **r** is a pattern, then:

```
r+
```

matches **one or more** occurrences of **r**. If there is a choice, then the longest leftmost string which matches **r** is chosen.

10. The concatenation of patterns is a pattern which matches the concatenation of the strings matched by each component of the pattern.

Output from **grep** may be redirected to a file by using:

```
>filename
```

as the first command-line argument to **grep**, where **filename** refers to the name of the desired output file.

Files to be searched may be specified to **grep** in any manner understood by AmigaDOS. The AmigaDOS wildcard patterns are enumerated in **The AmigaDOS Manual** under the **list** command and can be used to specify groups of files to be searched. Refer to Appendix C for details of **The AmigaDOS Manual.** For example:

df0:#?.c	Matches all files on drive df0: with a **.c** file extension.
g#?.c	Matches all files in the current directory with extension **.c**, beginning with **g**.

Any of the file specifications to **grep** can also be combined, such as:

#?.c s/#?.c	Matches all files both in the current directory and in the subdirectory **s/**, with a **.c** file extension.

SEE

grep Libraries.

NAME

lc Lattice C Compiler

SYNOPSIS

```
assign LC: executable path
assign INCLUDE: include path
assign LIB: library path
assign QUAD: quad path

lc options files
```

DESCRIPTION

The normal method for executing the Lattice C Compiler is to invoke the **lc** command. The *assign* statements in the synopsis are optional. The **lc** program separates the options list into those for pass 1 and those for pass 2. Then it executes **lc1** (pass 1) and **lc2** (pass 2) for each of the C source files specified by the files list. This list can consist of one or more file names and/or file patterns, separated by white space. For example:

```
lc #? :mydir/myprog :mydir/abc?
```

will compile all C source files in the current directory, plus the source file named **:mydir/myprog.c** plus all C source programs in the **:mydir** directory which have four-character names beginning with *abc*. Note that the **lc** command automatically supplies the *.C* extension on all source file names.

The LC command will also automatically invoke the librarian and linker if required.

This command utilizes the logical name assignment feature of AmigaDOS to locate the various programs and files. These assignments allow these programs and files to be located in any directory on any disk.

The options list can contain any combination of the following, separated by blanks:

-a This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory addressable by the Amiga's custom hardware. This is necessary for screen image and audio data. This option is synonymous with the **-c** option for **lc2**. The **-a** option must be immediately followed by one or more of the following letters in any order:

b	bss or uninitialized data
c	code segment
d	data segment

The default action is to load the segments into memory not addressable by the custom hardware if it is available. This is generally desirable because it avoids bus contention between the processor and the custom hardware. Image and audio data must be loaded into memory accessible by the custom hardware. For example:

```
-acdb
```

will cause all segments to be loaded into chip addressable memory, regardless of the system memory configuration.

Note that this option is superseded by the new chip keyword supported in LC1. When used with the LC program it will automatically specify -b0 to override the -b1 default. By changing existing code to use the chip keyword, you will be able to use the improved code quality offered by the -b1 option.

-b This option causes the compiler to change the form of addressing used to locate statics, externals and strings. By default, -b1 is used to imply that all such items are addressed as a 16 bit offset from address register A4. The disadvantage of this is that it only allows for 64K bytes of data that can be addressed and all such data must be in the same hunk. You can override this option by using the -b0 option which implies full 32 bit addressing for accessing all items.

Note that this option does not limit the amount of data that may be allocated at run time.

When using the -a option of LC, you must specify -b0 in order to access any statics and externs declared in a module compiled with the -a option. In order to eliminate the requirement for the -b0 option, you can utilize the chip keyword on those objects that are to be placed into chip memory.

This option now is passed to LC1 instead of LC2 where it actually causes the compiler to change the default storage class of externs to "far" or "near" as appropriate. If you have a program which has a large amount of data, you can readily use the -b1 default by putting the "far" keyword on any large objects to move them out of this common merged data section.

-C This option causes the LC command to continue with the next source file when a fatal compilation error is reported while multiple source files are being compiled. Normally, a fatal error causes the process to pause with the following message displayed on your screen:

```
Compiler return code xx.
Press Y to abort, any other key to continue.
```

The compiler error messages are also displayed immediately above this prompter. If you respond with a 'Y' (yes), LC will abort, otherwise it will proceed to the next source file.

-c The compiler defaults to compatibility with previous releases with many ANSI C language features, but the -c *compatibility option* can be used to activate some important features as well as compatibility with other compilers. The **-c** must be immediately followed by one or more letters from the following list, in any order. We recommend that you use the options **-cusf** for the best code generation and error reporting.

+	Compatibility mode for the Lattice C++ product. This will suppress warnings associated with structure passing and other potential problems areas that will have already been diagnosed by the C++ front end.
a	Enables full ANSI compatibility mode with full diagnostics to check for portability problems. Some features of the compiler are disabled when this option is specified such as precompiled header files and suppressing multiple includes of the same file in order to acheive compliance.
c	Allows comments to be nested.
d	Allows $ character to be used in identifiers.
e	Suppresses the printing of the error source line in conjunction with any warnings or errors.
f	Forces the compiler to check for the presence of function prototypes and to complain when one isn't present at a function call or function definition.
i	Suppresses multiple #includes of the same file. If a second #include of the same file is encountered, the directive is simply ignored. Note that case is important although no distinction is made for angle brackets or quotes. This option is implied when precompiled header files are used or created.
k	Enables the presence of the chip, near and far keywords even when the -ca option has been specified.
l	Use allows a pre-ANSI language dialect.
m	Allows use of multiple character constants (e.g. 'ab')
n	Allows nesting of #define symbols
o	Provides a compatibility mode to use the pre-ANSI style preprocessor found in previous releases of the compiler. The most important aspect of this occurs in substitution of symbols within quoted strings.

r Enables registerized parameter passing based on function prototypes and definitions.

s Causes the compiler to generate a single copy of all identical string constants into the code section of the program. Note that when this option is specified, modification of any string constants at runtime will produce unpredictable results.

t Enables warning messages for structure and union tags that are used without being defined. For example:

```
struct XYZ *p;
```

would not normally produce a warning message if structure tag XYZ were not defined.

u Forces all *char* declarations to be treated as *unsigned char*.

w Shuts off warning messages generated for return statements which do not specify a return value within an **int** function. By the ANSI standard, all such functions should be declared as **void** instead of **int**

x Causes all global data declarations to be treated as externals. This is identical to specifying the -x option.

-d This option has two uses. When used by itself or immediately followed by a numeric digit, it activates the debugging mode. Currently, the following debugging options are supported:

-d0 Disables all debugging information

-d1 Enables output of the line number/offset table

-d Same as -d1

-d2 Outputs full debugging information for only those symbols and structured referenced by the program.

-d3 Outputs full debugging information for only those symbols and structures referenced by the program. Additionally it will cause the code generator to flush all registers at line boundaries.

-d4 Outputs full debugging information for all symbols and structures declared in the program even if there is no reference to them.

-d5 Outputs full debugging information for all symbols and structures declared in the program even if there is no reference to them. Additionally it will cause the code generator to flush all registers at line boundaries.

When any of the debugging options is specified, the preprocessor symbol **DEBUG** will be defined so any debugging statements in the source file will be compiled.

The -d option can also be used to define preprocessor symbols in the following ways.

-dsymbol Causes **symbol** to be defined as if your source file had the statement:

```
#define symbol
```

-dsymbol=value Causes **symbol** to be defined as if your source file had the statement

```
#define symbol value
```

-e This option causes the compiler to recognize the extended character set used in Asian-language applications. Note that this option has no validity on the current generation of hardware running v1.2 or v1.3 of the system software.

-f This option controls the format to be used for all floating point operations. Currently four basic styles of floating point are supported:

-fl Standard Lattice IEEE routines linked into the program to perform software emulation of all floating point operations. This code will work on all machines but will not take advantage of a 68881 if present. This option is the default for compatibility with previous version, of the compiler.

-ff Mototola Fast Floating Point (FFP) format for all floating point operations. The resulting object module must be linked with the FFP math library **lcmffp.lib**. In order for code compiled with this option to run, the user must have **mathffp.library** in the libs: directory. While these routines are slightly faster than standard IEEE routines, they operate with much less precision. These routines will not benefit from the presence of a 68881.

-fi Amiga IEEE routines located at run time. In order for code compiled with this option to execute, the user must have **mathieedoubbas.library** in the libs: directory. While the overhead of calling the library is slightly greater for the software emulation, the presence of a 68881 will be detected by and used by the library supplied with the 68881 board. We recommend that you use this option for any math intensive application.

-f8 Inline Motorola 68881 generated instructions using the co-processor interface. Code compiled with this option will not operate unless a 68881 is installed which conforms to this interface. This generally implies the presence of a 68020/68030 processor as the 68000 does not support such an interface. While this provides the absolutely fastest form of floating point calculations, the number of target machines supported is quite limited.

One important note: If you are attempting to run two programs at the same time compiled with this option and they produce incorrect or inconsistent results, there may be a problem with the board not correctly hooking into the task

switch/ launch facilities. To date, we know that the Commodore supplied board does conform to such specifications.

In addition to the floating point styles, the compiler allows some control over the precision attributed to the *float* and *double* declarations used within the user code.

-fs Causes the compiler to treat all declaractions as single precision. This is the only mode allowed with the FFP format and is optional for all others.

-fd Causes the compiler to treat all declarations as double precision. This is illegal in conjunction with the FFP format.

-fm Causes the compiler to treat *float* as single precision and *double* as double precision. This option is the default for all formats except FFP (which is illegal with this option).

-f Will reset to the default of Lattice IEEE mixed mode. If you specify both a floating point style and a precision, it must be done on the same **-f** option such as in **-fim** or **-ffs**.

-g This option causes the big version of LC1 to generate a cross reference and listing file. LC will automatically invoke this version of the compiler if the option is specified. The -g option is followed by one or more of the following option letters in any order:

c Outputs a cross reference of all compiler provided include files found by searching the include: directory. By default these symbols are not printed.

d Includes all **#define** symbols in the cross reference listing.

e Causes the source listing to display all excluded lines as controlled by **#if** or **#ifdef**. Normally these lines are not displayed.

h Includes the contents of all include files found in the include: directory as they were included by the source program. Normally, only the #include directive causing the compiler to read the file is displayed.

i Includes the contents of all user provided include files in the expanded listing.

m Displays both the original source line and the line after macro expansion in the listing. This is useful for tracking down problems related to preprocessor replacement of symbols.

n Toggles the narrow mode of the listing. By default, the listing will be formatted for a 108 column line with most lines not exceeding 80 characters. When enabled, this option allows for listing lines up to 132 characters.

s Toggles listing of the input source code.

x Toggles generation of a cross reference of the symbols encountered in the source file.

-h This option specifies that the sections designated by the characters which immediately follow are to be loaded into memory not addressable by the Amiga's custom hardware. The **-h** option must be immediately followed by one or more of the following letters in any order:

b bss or uninitialized data

c code segment

d data segment

The default action is to load the segments into memory not addressable by the custom hardware if possible. This option prevents the specified segments from being loaded into chip addressable memory even if no other memory is available. This can cause programs to not execute if external high-speed memory is not available on the machine. As an example:

```
-hcdb
```

would cause all three sections to be loaded only in high-speed ram.

-H This option specifies that the compiler is to preload the symbol table from a precompiled header file. It is immediately followed by the name of the precompiled header file as in:

```
-Hinclude:all.sym  -Hall.sym
```

There is no limit to the number of precompiled header files that may be read in.

-i This option specifies a directory that the compiler should search when it is attempting to find an include file. For example, if you specify the option as **-idf0:headers -idf1:local** and then place the line:

```
#include "defs.h"
```

in your source program, the compiler first tries to find the header file names **defs.h** in the current directory. If it is not there, then the compiler searches for **df0:headers/defs.h** and **df1:local/defs.h** in this order. Finally, if these attempts fail, the compiler will attempt to open **include:defs.h**.

Note that you can place up to 16 -i options on the command line.

If the file had been enclosed in angle brackets:

```
#include <defs.h>
```

Then the compiler will attempt to open **include:defs.h** followed by **defs.h** in the current directory, followed by the -i specifications

-j This option allows control over the error messages reported by the compiler. It is immediately followed by a number and then an optional letter:

-j<n>	Causes the compiler to suppress printing of warning number <n>
-j<n>e	Causes the compiler to treat any occurences of warning <n> as an error instead.
-j<n>i	Causes the compiler to suppress printing of warning number <n>.
-j<n>w	Enables printing of warning <n>. By default, several ANSI oriented messages are disabled.

Several messages may be affected with the same -j option such as **-j22i30e132w** which disables warning message #22, turns #30 into an error and enables #132 as a warning.

-L When this option is present, **lc** invokes the linker if all compilations are successful. The first source file name is used as the name of the executable and map files produced by the linker. Any other files that were compiled are supplied to the linker as secondary object files. The Lattice C startup routine is included as the first object module.

The linker is instructed to search the standard Lattice library file **lc.lib** and the standard Amiga library file **amiga.lib**.

Additional Lattice libraries and linker options may be specified by immediately following the **-L** option with one or more of the following letters:

a This invokes the **ADDSYM** option of the linker. It causes debugging information for all routines to be output the executable file.

n This invokes the **NODEBUG** option of the linker. It causes all debugging information to be stripped from the final executable.

t This selects the typical linker options we recommend for producing small code. It currently selects **SMALLCODE SMALLDATA NODEBUG**.

c This invokes the SMALLCODE option of the linker. It causes all code hunks to be combined into a single code hunk.

d This invokes the SMALLDATA option of the linker. It causes all data and bss hunks to be combined into the single data hunk that is referenced by modules compiled with the **-b** option.

f This letter specifies that the Motorola FFP math library **lcmffp.lib** is to be searched before the standard run-time support library.

h This letter directs the linker to output the hunk portion of the map. This is the default map if no other map options are specified.

l This letter directs the linker to include library information in the map file.

m This letter specifies that the Lattice IEEE math library **lcm.lib** is to be searched before the standard run-time support library.

o This directs the linker to include overlay information in the map file.

s This letter directs the linker to produce a symbol listing in the map file.

v This invokes the VERBOSE option of the linker. It causes the linker to display statistical messages as it is processing the object files and libraries.

x This directs the linker to include cross reference information in the map file.

For example, **-Lm** will search **lcm.lib** before **lc.lib** and **amiga.lib**, and **-Lvf** will search **lcmffp.lib, lc.lib**, and **amiga.lib**, and display messages regarding the current linker status. Note that the math library options are mutually exclusive, and that the standard libraries are always searched last.

If you want to search other libraries, you must list those libraries after the letters options, and use plus signs as separators. For example, **-L+myfuncs.lib** searches **myfuncs.lib** before the standard Lattice and Amiga libraries, while **-Lm+myfuncs.lib+:george/myfuncs.lib** searches the libraries **myfuncs.lib, :george/myfuncs.lib, lcm.lib, lc.lib**, and **amiga.lib**. Note that the special libraries are searched before the Lattice libraries.

The **-L** option creates a file in the current directory named **xxx.lnk**, where **xxx** is the name of the first source file to be compiled (i.e., the same name that is used for the executable and map files). This *.LNK* file serves as input to the linker, and it is not deleted at the end of the procedure. This allows you to easily re-link if, during your testing, you find a need to change and re-compile only one module. To do this, simply execute the linker utility **blink** in the following way:

```
blink WITH xxx.lnk
```

where **xxx.lnk** is the name of the .LNK file previously produced by the **lc** command.

-l This option causes all objects except characters, short integers, and structures that contain only characters and short integers to be aligned on longword boundaries (i.e., addresses evenly divisible by 4). Structures will be longword aligned if they contain any members that must be aligned.

Note that this option is incompatible with any of the Amiga Exec structures which do not conform to this alignment requirements.

Attempting to call an Amiga system routine with such misaligned structures will produce unpredictable results and a very likely visit to the GURU.

-M When this option is present, **lc** will compile only those source files with dates more recent than the corresponding object files. Note that the dates of included files are not checked. In other words, if you change a header file without changing the source file that includes it, the source file will not automatically re-compile because it still pre-dates its object file.

For larger projects where there is a more intense dependency upon structures in common data files being changed, we recommend using a make utility such as **LMK** to manage recompilation of the affected source files automatically

-m This option allows control of the type of code generated. The -m must be immediately followed by one or more letters from the following list in any order:

0 Causes the compiler to generate code which will run on a Motorola 68000 found in most Amigas. Decisions on code optimization will be based on the timings for this processor.

1 Causes the compiler to generate code which will run on a Motorola 68010. Decisions on code optimization will be based on the timings for this processor. In general, code for this will run on a 68000 although the 68010 has instructions not found on the 68000.

2 Causes the compiler to generate code optimized for the 68020 processor. This code will not run on a 68010 or 68000 although it will run on a 68030.

3 Causes the compiler to generate code optimized for the 68030 processor. This code will not run on a 68010 or 68000 although it will work on a 68020.

a Causes the compiler to generate code to run on any Motorola 680x0 family processor. Code is optimized for the 68010/68020 only when it does degrade performance for a 68000.

c Disables the CORE deferred stack cleanup optimization which leaves parameters on the stack after a call to be reused and cleaned up by a subsequent subroutine call or return statement.

r Disables the automatic registerization of variables. By default, the compiler will attempt to pick likely candidates for register variables.

s Causes the compiler to choose optimizations which result in a reduction of space instead of time.

t Causes the compiler to choose optimizations which result in a performance increase at the cost of code space. This is the default.

-n This option causes the compiler to retain only 8 characters for all identifiers. The default maximum identifier length is 31 characters. In either case, anything beyond the maximum length is ignored.

-O This option invokes the global optimizer.

-o This option should be followed by the drive, directory, or complete file name for the object file that is produced by pass 2. Several examples are:

-odf0: Places the object file in the root directory on drive df0:.

-o:obj/ Places the object file into directory **obj/** on the current drive. The name of the file is the same as the source file name, with a .O extension instead of .C.

-ospecial.o Places the object file into the current directory with the name **special.o**.

-p This option is used when using the compiler in a preprocessor mode to produce a file used by subsequent compiler invocations. When this option is used, the compiler will not create a quad file. However, the file specified as the -o option will be used as the target name for the created file. There are several uses for the -p option:

-p by itself causes the compiler to write the results of preprocessing the input source file into the output file. If no output file is specified, a file extension of .p will be used to create the file.

-ph Causes the compiler to generate a precompiled header file containing a dump of all symbols encountered in the given source file. This file may then be used for the -H option on subsequent compiler invocations to reduce compilation time.

-pr Causes the compiler to generate a prototype file containing prototypes for all functions defined in the source file. The -pr may be immediately followed by one or more of the following option letters in any order:

e Eliminates prototypes for all static functions. Only those functions available externally will have prototypes generated for them.

p Causes the compiler to generate prototypes with **__PARMS** for portability to other compilers.

s Generates prototypes for all static functions. Only those functions defined with the static function will be output.

Note that -pres will not generate any prototypes.

-q This option has two uses. If the -q is immediately followed by a letter, it specifies where the quad file is to be generated. Otherwise it is used to control how many errors/warnings will be allowed before quitting a compilation.

This option should be followed by the drive, directory, or complete file name for the *quad file*, which is the intermediate file generated by pass 1 and read by pass 2. Several examples are:

-qram:	Places the quad file in ram disk.
-qdf0:	Places the quad file in the root directory of drive df0:.
-q:quad/	Places the quad file into directory **quad/** on the current drive. The name of the file is the same as the source file name, with a .Q extension instead of .C.

Note that the quad file is automatically deleted after pass 1. If you do not want this to happen, call the **lc1** command directly.

To control the maximum number of errors/warnings, the -q should be immediately followed by a number then either an e or w. For example:

-q3w	Quit after 3 warnings or errors.
-q2e	Quit after 2 errors.
-q10w1e	Quit after 10 warnings or any errors.
-q	Quit after any errors or warnings.

Note that when the compiler quits due to too many errors/warnings, it will not generate a quad file.

-r This option is used to control how the compiler is to generate subroutine calls and entries. The -r option may be followed by one or more of the following characters in any order:

0 Defaults all subroutine calls to "far" which means that the compiler will use an absolute 32-bit relocated address to locate the target function. Note that any functions explicitely declared "near" will use the more efficient 16-bit relative offset.

1 (The compiler default) Causes all subroutine calls to be defaulted to "near" which means that the compiler will use a 16-bit PC relative address to locate the target function. In order for this to work, the target subroutine must be within +/-32K of the generated instruction. If it is not within range, the linker will generate an ALV to allow the call to be bridged to the final target. Any functions explicitly declared "far" will use the larger 32-bit address.

r Causes the compiler to use registerized parameters for all subroutine calls and entry points. The first two integral and two pointer items will be loaded into d0-d1/a0-a1 for the call. Any function without a prototype or explicitly declared **_stdargs** will use the normal stack conventions.

s (The compiler default) Causes the compiler to use standard stack parameters for all subroutine calls. Those functions explicitly declared **_regargs** will use the registerized parameter conventions.

b Defaults the compiler to use registerized parameters for all subroutine calls, yet still generate a prologue that handles both styles of parameter passing.

-R When this option is specified, the object modules produced by the compiler are automatically inserted into a library file, replacing modules of the same names. The option must be followed by the name of the library, as in

```
-Rmylib.lib
```

which places the object modules into the **mylib.lib** library file. The **-R** can be followed by any valid file name, including drive code and path. A .lib extension is not automatically supplied.

-s This option allows you to change the default hunk or segment names generated by the compiler. If it is not present, the hunks are unnamed. You can use this option to provide hunk names as follows:

-s	This causes the compiler to use the default names of **text** for the program section, **data** for the data section, and **udata** for the bss or uninitialized data section.
-sc=codename	Causes the compiler to use the name **codename** for the program, or code, section without affecting the names of the other sections.
-sd=dataname	Causes the compiler to use the name **dataname** for the data section without affecting the names of the other sections.
-sb=bssname	Causes the compiler to use the name **bssname** for the bss, or uninitialized data, section without affecting the names of the other sections.

-u This option by itself undefines all preprocessor symbols which are normally pre-defined by the compiler.

-v This option disables the generation of stack checking code at the beginning of each function.

-w This option causes the compiler to treat all integers as 16-bit short values. It is intended to provide compatibility with other compilers although it does provide a marginal increase in per-

formance of the generated code. When using this option, we strongly recommend use of prototypes to catch parameter mismatch errors as not all parameters will be promoted to 4 bytes as is the default.

-x This option causes all global data declarations to be treated as externals. This can be useful if you define data in a header file that is included by multiple source files. The **-x** option can be used with all the files except one, in this case, to cause the data items to be defined in one module and referenced as externals in the others.

-y This option causes each function entry sequence to load address register A4 with the value of the linker defined symbol **LinkerDB**. This symbol is the data section base address, biased as necessary. This option must be used if the **-b1** option is used with interrupt code or functions that will be multi-tasked with the **AddTask()** function. Note that in general only the functions that can be used as entry points to the task or interrupt handler need to use this feature, since register A4 will be propagated by subsequent function calls.

This option is superseded by the **__saveds** option keyword that may be used with a function. Any function having this keyword will automatically load up the base register upon entry.

RETURNS

The LC command returns the following completion codes:

0 All compilations were successful. That is, at least one source program was compiled, and there were no fatal errors.

1 One or more fatal compilation errors were reported.

2 No source files were found.

SEE

lc1, lc2, blink

EXAMPLE

The first command assigns the INCLUDE logical name so that header files will be found in the **df0:include** directory. The second command assigns the LC logical name so that the driver program can find the executables in the **df0:c** directory. The next command assigns the LIB logical name so that the driver program can tell the linker to locate the standard library files in the **df0:lib** directory. The next command compiles all modules in the **mylib** subdirectory and then constructs a library named **mylib.lib** if all compilations are successful. The last command compiles **myprog.c** and links it with **mylib.lib** to produce a load module named **myprog**.

```
lc  -Rmylib mylib/#?
lc -mp -L+mylib myprog
```

NAME

lc1	Compiler pass 1
lc1b	Big comiler pass 1

SYNOPSIS

```
lc1 options file
lc1b options file
```

DESCRIPTION

lc1 and **lc1b** commands invoke the first compiler pass, which reads a source file and translates it into an intermediate form known as a *quad file*. **lc1b** invokes the big compiler for cross referencing and prototyping purposes. **lc1b** will be invoked automatically if the -g option and the prototyping options are used. Unlike the **lc** command, you can only specify one source file to **lc1**, and it should be written without the .C extension. For example, if the file argument is **myprog**, this pass will translate **myprog.c** into the quad file named **myprog.q**.

The **options** argument can consist of the following items, which are described in the *lc* command:

-b	Base register relative data addressing.
-c	Compiler compatibility settings.
-d	Debugging mode or preprocessor symbol definition.
-e	Extended character set processing.
-f	Floating point format selection.
-g	Listing generation options. This is only valid with the **lc1b** command.
-h	Precompiled header file inclusion
-i	Directory paths for local include files.

-j	Error/warning message control
-l	Longword alignment of data items.
-n	Long symbol names.
-o	Specify destination for .Q file. Note that this option is specified as **-q** on the *lc* command.
-p	Preprocessor options.
-q	Compilation error abort control.
-r	Subroutine call control. This option was an option for the *lc2* command in previous versions of the compiler.
-u	Undefine preprocessor symbols.
-w	Generate code to use short integers.
-x	Treat all global declarations as externals

All options must appear before the file name.

SEE

lc, lc2

NAME

lc2 Compiler code generator

SYNOPSIS

```
lc2 options file
```

DESCRIPTION

This command invokes the second compiler pass, which reads a quad file and translates it into an object file. Note that you can only specify one quad file to **lc2**, and it should be written without the .Q extension. For example, if the file argument is **myprog**, this pass will translate **myprog.q** into **myprog.o**.

The **options** argument can consist of the following items, which are described in the **lc** command:

-c	Chip memory specification. This is the same as the **-a** option on the **lc** command.
-h	Fast memory specification.
-o	Specify destination for .O file.
-s	Specify segment name.
-v	Disable stack checking.
-y	Unconditionally load the base register

All options must appear before the file name.

SEE

lc, lc1

NAME

lcompact Header file compressor

SYNOPSIS

```
lcompact <infile >outfile
```

DESCRIPTION

This command is used to create a compressed version of a header file that may be processed by the compiler faster.

By default, **lcompact** reads from the standard input file and writes to standard output. To cause it to read from a given file, you must use the AmigaDOS redirection character **<** as in **<infile**. If you do not do this, it will read all input from the CLI window until an EOF ^\ is encountered.

The result may be placed into a file by redirecting the standard output with the AmigaDOS redirection character **>** as in **>outfile**. If you do not specify a redirection, the output will be displayed on the CLI window. Since this is binary coded data, the output will not be very intelligible.

The **lcompact** command compresses a file by performing four basic operations:

Compression	Multiple blanks and meaningless blank characters are removed. Expressions are analyzed according to standard C precedence rules to determine which are safe to remove.
Elimination	All comments and unnecessary blank lines are removed.
Transformation	Certain sequences such as hex constants are converted into more efficient representations. For example, *0x0000001* is transformed to *1*. These transformations take place only if the secondary representation is more space efficient.

Tokenization	A limited number of common keywords are reduced to a single token character with the high order bit set. This fixed set of keywords is known to the compiler and will be automatically expanded as they are encountered.

The end result of compacting a header file is anywhere between 20% and 75% smaller depending upon the original contents. The compiler will automatically expand the header file on input so that error messages will print out the original line (without comments) for a diagnostic.

SEE

lc1

NAME

lmk Maintain and update records of file dependencies

SYNOPSIS

```
lmk [options] [macro definitions] [targets]
```

DESCRIPTION

lmk, a descendant of the UNIX **make** utility, is a tool for maintaining file dependency relations and aids in performing the minimal necessary actions in order to remake specified targets.

The basic operation of **lmk** is to:

- Locate and identify the target file in the lmk or description file.
- Ensure that any file on which the target file depends, including any file needed to create the target file, already exist and are up to date.
- Create the target file if any of the dependent files have been modified more recently than the target file.

In this context, a basic description of these terms is:

dependent file
: This file is the basic unit of the **lmk** structure. Such a file could be a source code file, header file or object code file. The dependent file is the file needed to create a target file, i.e. if a dependent file is changed (for example, by editing), then the target file must be rebuilt.

lmk file
: This file contains the names of every file which are required to create the target file. This file also contains details of the actions necessary to create the target file. These actions can be such things as directory locations, compile or linking sequences of the dependent files, as well as basic housekeeping commands.

target file — This is the file which is the end result of an **lmk** session. In other words, any file that is created or updated by **lmk**.

Command line options, or flags, can be entered discretely or in conjunction:

```
lmk -pn
```

and

```
lmk -p -n
```

are both allowed. Flags may be specified in any order provided a flag is placed adjacent to its parameters, if any. The following summarizes all of the available command line options for **lmk**.

-a — This option will rebuild all targets without regard to time stamps. This forces all targets and sub-targets to be rebuilt.

-b file — This option will use the filename following this flag as the default file, rather than the file named **lmk.def**.

-c — This option will create a batch file from this **lmk** session. The batch file will be passed for execution with the command **execute lmkfile.bat**. The file **lmkfile.bat** will be left in the current directory giving a history of the actions and responses of the **lmk** session.

-d — This option will print out detailed debugging information about the processing of the **lmkfile**.

-e — This option will erase any out-of-date targets before remaking them.

-f name — This option will use the filename following this flag as the input **lmkfile**.

-h — This option will print out help information and then exit.

-i This option will ignore error returns from actions.

-k This option will ignore error returns from actions passed to AmigaDOS. Also, ignores error returns stemming from **lmk** not knowing how to make certain targets.

-n This option will display the actions **lmk** would have taken onto the screen, but will not execute these actions.

-p This option will print out target descriptions and expanded macros.

-q This option will query if the **target** file is updated. Prints a 1 if the target is currently up to date or a 0 if it is not. **lmk** will not take any of the associated actions.

-s This option will not echo actions to the screen before executing them.

-t This option will **Touch** the target files by updating them with the current system time. None of the actions usually associated with making these targets will be performed.

-u This option will rebuild unconditionally all targets without regard to time stamps. This will force everything (all targets and sub-targets) to be rebuilt.

-x This option is the UNIX compatibility flag.

lmk is based upon a rule structure which can be extended and customized. **lmk** allows the substitution of repetitive character strings into macros and provides internally defined or default macros. These macros are subject to certain rules within **lmk**. Refer to the section dealing with **lmk** for full details of macros and the rule structure. In addition, there are certain characters which are part of the rule structure of **lmk**, or have a special meaning. The special characters are:

The minus symbol (-) causes **lmk** to continue processing even after an error condition has been encountered.

@	The at symbol (@) causes **lmk** to echo actions rather than command lines and resulting actions to the output device.
~	The tilde symbol (~) is used as an escape symbol for any special character used on a command line.
&	The ampersand symbol (&) is meant for source code compatibility with previous implementations of **lmk**.

lmk supports macros for representing directory specifications, header files and compiler or assembler flags. Any AmigaDOS command can be made into a macro. A macro is defined by use of the equals symbol (=) and takes the following form:

```
MACRO=DEFINITION
```

The macro is referenced within an **lmkfile** by the attachment of a dollar symbol ($) to the front of the macro name. The name must be enclosed in parenthesis. For example:

```
$(MACRO)
```

There are five default macros provided by **lmk** and these are recognized by the position of the dollar sign ($). The default macros are:

$@	Set to the target filename
$?	Set to a dependency list based upon files younger than the target file.
$*	Set to the prefix and basename of the first dependent filename.
$<	Set to the dependent filename which caused the actions.
$>	Set to the basename of the dependent file which caused the actions.

It is possible to override the definition of a macro from the command line of **lmk**. The reassignment of the macro is done by the use of the equals symbol in the form:

```
MACRO1=NEWTEXT
```

Note that no spaces are allowed between the name of the macro and the equals symbol. For example:

```
LMK DEBUG=YES -f editor
```

Transformation or inference rules inform **lmk** how to make a file with a given filename extension from a dependent with a given filename extension. In other words, this means that the rule to convert a **.a** file into a **.b** file would be termed **.a.b**. When used in an **lmkfile**, the transformation rule would be followed by a colon, e.g. **.a.b:**.

Three default macros are set whenever a transformation rule is used:

$* is set to the prefix of the <u>first</u> dependency filename including any directory paths. In other words the filename extension is ignored.

$< is set to the dependency filenames which are out of date.

$> is set to the basename of the dependency filenames which are out of date.

Comments may be placed within an **lmkfile** by use of the hash symbol (#). For example:

```
#  This line is a comment
```

Rules internal to this implementation of **lmk** assume you are using the **Lattice AmigaDOS C Compiler**. These rules are taken from the file named **lmk.def** on the distribution disk. You are able to override these rules if desired, by editing the **lmk.def** file.

```
LC = LC:lc
LC1 = LC:lc1
LC1FLAGS = -iINCLUDE: -iINCLUDE:lattice/
LC2 = LC:lc2
AS = LC:ASM
LINK = BLINK:blink
```

```
.DEFAULT:
   $(LC)  $(LC1FLAGS) $*

.a.o:
   $(AS)  $* -i Include:/Assembler_Includes -o  $*.o

.c.o:
   $(LC)  $(LC1FLAGS)  $*

.h.o:
   delete  $*.o
   $(LC)  $(LC1FLAGS)  $*
```

lmk provides support for a number of special or *fake* targets. In other words, an expression which appears to be a target in an **lmkfile** is actually used to control the operation of **lmk**. Fake target names must always begin with a period (.) and be capitalized:

- .DEFAULT
- .IGNORE
- .ONERROR
- .SET
- .SILENT

The results of each *fake* targets are tabulated below:

Fake Target	Result
.DEFAULT	Specify default rules
.IGNORE	Same as -i option
.ONERROR	Specify error response
.SET	Set logical assignment
.SILENT	Same as -s option

lmk supports the use of local input files. Local input files provide a method of keeping actions associated with making a target in the same **lmkfile** rather than in a separate file on disk. Any macros defined in the **lmkfile** can be used within a local input file.

Commands to appear within the temporary file are enclosed between the two delimiter characters of < and <.

A template for construction is:

```
command <[preface]<[!] [ (filename) ]
    stmt1
    stmtN
    <
```

where:

command The action to be handed to AmigaDOS.

preface The symbol which is to precede the local input filename when it is given to AmigaDOS.

! The instruction to **lmk** to create a a local input file containing the statements specified in the rest of the command line.

filename The optional filename given to the local input file created.

stmt1-N The statements within the local input file.

NAME

lprof Execution Profiler

SYNOPSIS

```
lprof command
```

DESCRIPTION

The **lprof** command is used to gather information about where a program is spending most of its execution time.

In order to gather statistics on a given command, prefix the command with the **lprof** command. Note that if the command you wish is not in the current directory, you must fully specify the command location even if it is in the path. Also, any redirection operators which were immediately after the command must now be moved to after the **lprof** command.

After the profiler loads the given command, it will execute it and gather statistics by examining the current program counter at given intervals. You can verify that the profiling is in progress by the power light fluctuating in intensity. When the program run completes, the **lprof** command will create a file **prof.out** in the current directory that you can then use with the **lstat** command.

SEE

lstat, lc, lc1

EXAMPLE

Given a simple program **foo** that doesn't take any arguments, you can gather statistics on it with:

```
lprof foo
```

For a program that normally takes command line arguments, you can add them after the command name as in:

```
lprof mycmd -opt 1 -flag
```

If the program needed to have output redirected such as in:

```
myprog >outfile -opt 2 -doit
```

You would instead move the redirection after the **lprof** command as in:

```
lprof >outfile myprog -opt 2 -doit
```

If the program were in the C: directory or another directory automatically searched by the CLI, you would need to give the full path name as in:

```
lprof C:mynewcmd -opt 2
```

NAME

lse Edit user files

SYNOPSIS

```
lse [-v | [[-ddatafile] file1[.ext] [file2[.ext]]]
```

DESCRIPTION

The **lse** (Lattice Screen Editor) is a full-screen integrated editor for the Amiga. The **lse** command line options are as follows:

-v This option displays the **LSE** version number and copyright. You will need the version number from this display should you call for support.

-ddatafile The option *-d* indicates that a datafile name is being entered. The datafile is the name of the file created by the installation program **LSEINST.EXE**. The default is to search the directories in the PATH variable for **LSE.DAT**. There must be no blank characters between the option *-d* and the datafile name.

file1[.ext] The file to be read into window 1. The extension is optional. Using **LSEINST** you can choose to automatically append default extensions to the file name (like **.c** for C source files).

file2[.ext] The file to be optionally read into window 2. Again, the extension is optional and appended only if the default (no extension) has been changed.

You can invoke **LSE** without a filename by entering:

```
lse ↵
```

Use the *Project Menu* and the *Rename Option* to name a text file invoked without a filename. **LSE** will not allow you to save an un-named text file.

NAME

lstat Profile reader and printer

SYNOPSIS

```
lstat >output options program profile
```

DESCRIPTION

This command is used to analyze a profile statistics file created by the **lprof** command.

The **program** field is required and indicates the name of the executable for which the statistics were gathered. **lstat** uses this to obtain all the debugging and symbol information for its report. The **profile** field is optional and is used to override the default of **lprof.out** which is written by the **lprof** command. You will wish to use this if the file is in a different directory, or you have renamed the file.

The **>output** field is optional and redirects **lstat's** output from the screen to an output device or file. Most programmers use **lstat** by redirecting its output to a file and then printing the file. See the examples below.

The **options** field need not be present. **lstat** accepts the following options

-z This option instructs **lstat** to display statistics for all subroutines even if they were not encountered in profiling. The default is to suppress all entries that have a 0 hit count.

-f This option instructs **lstat** to display full statistics for each subroutine indicating the line numbers within a module that were hit by profiling. By default, only summary information about the subroutine is printed.

-t=**n** This option allows pruning of the information presented. By default, **lstat** will print all subroutines which had even a single hit. If you specify **n** it will only display those which have at least **n** or more hits.

Note that this command assumes that you have compiled with at least **-d1** debugging from **lc1** in order to be able to associate the code with the given

line. In the absence of this information, the **lstat** command will only report statistics on a subroutine by subroutine basis.

SEE

lprof, lc, lc1

EXAMPLE

These examples assume that the file **lprof.out** is in the current directory. It is created by running the command:

```
lprof testprog
```

To simply view the basic statistics about the run, we can give the command:

```
1> lstat testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                      Most hits -  1.2121% (8) on line 80
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                      Most hits -  1.2121% (8) on line 132

                ...
  0.1515% 100.0000%  1ca18        1  CXM22
```

We can get an indication of the additional routines that did not get encountered by the profiler with the **-z** option. Note the extra routines at the end which have 0 hits and account for 0.0000% of the execution time that now appear in the output.

```
1> lstat -z testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
```

```
                                      Most hits -  1.2121% (8) on line 80
    2.2727%  74.3939%  18252      15  instal [lines 117-146 in sym.c]
                                      Most hits -  1.2121% (8) on line 132
              ...
    0.0000% 100.0000%  1ca60       0  strcat
    0.0000% 100.0000%  1ca78       0  strncpy
```

More information about the individual lines in a subroutine may be obtained with the **-f** option. For each routine that has line number information, it will indicate the percentage of total execution that was spent on that line in the subroutine. Note that subroutines without line number information will not show any additional information.

```
1> lstat -f testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661

Routine %   Total % Offset     Hits   Label
 59.5455%  59.5455%  1c44c      393  getrsc
  8.9394%  68.4848%  1c6fa       59  frersc
  3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                        0.3030% (2) on line 64
                                        1.0606% (7) on line 68
                                        0.7576% (5) on line 69
                                        1.2121% (8) on line 80
                                        0.3030% (2) on line 90
  2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                        0.1515% (1) on line 117
                                        0.1515% (1) on line 129
                                        0.4545% (3) on line 131
                                        1.2121% (8) on line 132
                                        0.1515% (1) on line 133
                                        0.1515% (1) on line 138
            ...
  0.1515% 100.0000%  1ca18        1  CXM22
```

With a large program, the amount of information presented by the profiler can be overwhelming. In attempting to improve performance of such a program, it is beneficial to limit the amount of information that the **lstat** program presents. The **-t** option can be used to set a threshold below which no information will be reported. With this option, the final total will not be 100%.

```
1> lstat -f -t=2 testprog
header size 0x1c, ProfileHeader 0x1c
719 run + 158 ready + 77 wait = 954
 75.367% run,  16.562% ready,   8.071% wait
  8.067% samples (58) out of profile range

counted = 660, RUN - NonTable 661
```

```
 Routine %   Total % Offset     Hits   Label
  59.5455%  59.5455%  1c44c      393  getrsc
   8.9394%  68.4848%  1c6fa       59  frersc
   3.6364%  72.1212%  10d38       24  alcmem [lines 64-102 in memory.c]
                                         0.3030% (2) on line 64
                                         1.0606% (7) on line 68
                                         0.7576% (5) on line 69
                                         1.2121% (8) on line 80
                                         0.3030% (2) on line 90
   2.2727%  74.3939%  18252       15  instal [lines 117-146 in sym.c]
                                         0.1515% (1) on line 117
                                         0.1515% (1) on line 129
                                         0.4545% (3) on line 131
                                         1.2121% (8) on line 132
                                         0.1515% (1) on line 133
                                         0.1515% (1) on line 138
                     ...
   0.3030%  98.3333%  1ca00        2   xcovf
```

NAME

omd Object Module Disassembler

SYNOPSIS

```
omd >output options object source
```

DESCRIPTION

This utility program disassembles an object file produced by the Lattice C Compiler and produces an output listing consisting of assembly-language statements, possibly interspersed with the original C source code.

The **object** field is required and gives the complete object file name. That is, OMD will not automatically supply the .o extension. The **source** field is optional. If present, it must be the complete source file name. When this field is used, you should have compiled the source file with the **-d** option (see the *lc* command) to produce the debugging information in the object module that allows *omd* to associate a particular source line with the object code that was generated. If you did not use the **-d** option, then the C source lines will not appear in *omd's* output.

The >*output* field is optional and redirects *omd's* output from the screen to an output device or file. Most programmers use *omd* by redirecting its output to a file and then printing the file. See the example below.

The **options** field need not be present. The Amiga implementation of *omd* only accepts the following option:

-x — This option overrides the default size of the buffer used to hold the external symbol section of the object module. For example, **-x200** establishes a buffer that can hold 200 external symbols, which is the default. You should increase this value if *omd* reports that there are too many external symbols.

EXAMPLE

This example compiles *MYPROG.C* to produce *MYPROG.O*, which is then disassembled. The disassembled listing is saved in the file *MYPROG.LST*.

```
lc -d  myprog
omd >myprog.lst myprog.o myprog.c
```

NAME

oml Object Module Librarian

SYNOPSIS

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

DESCRIPTION

The object module librarian *oml* can create a library file by combining object modules, generate a listing of the modules (and their public symbols) contained in a library, or manipulate modules within an existing library file.

Library files provide a convenient way of collecting object modules to be presented as a group of available components during program linking; the load module builder then includes only those modules from the library which are actually needed by the load module being built. Libraries are especially useful when several programs make use of common subroutines, since these subroutines can be placed in a library file and included on an *as required* basis when the programs are linked.

A library file is made up of object modules, each of which was originally a single file. Each module within the library is identified by a module name, which is normally obtained from the object module itself (the Amiga object module format defines a special *program unit* or module name record). This name is placed in the object module by the translator (assembler or compiler) program which generates it. Some modules may not contain a module name at all; in that case, the object module librarian assigns a module name of the form **$nnn**, where **nnn** is a decimal number indicating that the module was the **nth** unnamed module encountered in the library.

In order to perform replacement of modules within a library file, it is necessary to insure that the module contains a program unit or module name. The Lattice AmigaDOS C Compiler and AmigaDOS assembler always place a module name in the object files they produce. The current versions of the

compiler and assembler use the name of the object file. Thus, compiling **ftoc.c** produces an object module with the name **ftoc.o**.

The Metacomco Amiga assembler does not place any module name in the output file unless the IDNT directive is used. For example:

```
        IDNT     "asmod.o"
```

causes the module name **asmod.o** to be placed in the object file created when the source file is assembled.

When the linker examines a library file to find modules to be incorporated into a program, the module name is not important; instead, the linker decides if a module is needed on the basis of the public symbols it defines. A module may define one or more such symbols, which identify program components such as functions or data elements. Because the presence of more than one definition for a symbol may cause confusion, the object module librarian warns when it examines or constructs a library file which includes multiple definitions of a symbol.

Each invocation of *oml* specifies a particular library file upon which operations are to be performed. Then, a sequence of one or more commands is used to indicate the desired operations.

Commands may be specified on the command line used to execute *oml*, or they may be read from **stdin**, or a combination of both kinds of input may be used. If no commands are specified on the command line which executed *oml*, they are automatically read from **stdin**, which is usually the user's console but can be redirected to a file. The special command @ (valid only on the command line) is used to switch command input from the command line to **stdin**; an explicit file name may be attached to the @ to force commands to be read from that file. Commands are read from a file or from **stdin** until an end of file condition is detected; if commands are being read from the user's console, a control-backslash must be used to end command input.

Each command is specified as a single character, usually followed by a list of module names or object file names. Commands and file/module names are separated from each other by white space. If a command is followed by one

or more names, the first name specified is NOT checked as a possible command; thus, names which might be confused with commands must be specified as the FIRST name following a command.

The format of the command to invoke the object module librarian is:

```
oml [<cmdfile] [>listfile] [options] libfile [commands]
```

The various command line specifiers are shown in the order they must appear in the command. Optional specifiers are shown enclosed in brackets.

<cmdfile. Causes commands to be read from the named file, provided that (1) no commands are specified on the command line or (2) the @ command (see below) is used to force commands to be read from **stdin**. If this option is omitted and neither of the above conditions is met, commands are read from the user's console.

>listfile. Causes the listing output generated by *oml* to be written to the named file. If omitted, listing output is directed to the console.

options. Librarian options are specified as a hyphen followed by a string of characters which may not include white space. Current options include the following:

-oprefix — Specifies that the output filenames for the **x** command are to be formed by prepending the module name with **prefix**. Note that if a directory name is to be specified as a prefix, a trailing node separator (a slash under AmigaDOS) must be supplied on the prefix.

-s — Causes a listing of the public symbols defined in the module to be included in the listing produced by the **l** command.

libfile. Specifies the name of the library file to be created or manipulated; this is the only command line field which must be present.

commands. These specify the actions to be performed by *oml* with respect to the specified library file. Commands are specified by a single character; if alphabetic, either upper or lower case may be used. They are separated from

each other and from elements of file or module name lists by any white space. Current commands include:

r file-list — Replaces the named object files in the library, or adds them to the library if not already present. Note that replacement of existing modules in a library will work correctly only if the file name is the same as the module name. If the module does not exist in the library, this is unnecessary.

d module-list — Deletes the named modules from the library. Since modules without program unit names are assigned module names by the librarian, it may be necessary to obtain a listing (via the **l** command) in order to determine the assigned **$nnn** name for the module which is to be deleted.

x module-list — Extracts the named modules from the library, creating files of the same names. Note that if the module name includes a path name, the librarian will attempt to create a file with that name. All files are created in the current directory, unless the *-o* option is used; in that case, each module name is prefixed with the text specified on the *-o* option. The special character '*' may be used to indicate that all of the modules in the library are to be extracted. *oml* will terminate execution if an attempt to extract a module is unsuccessful. Note that it is an error to specify the same name in both a replacement and an extraction list.

l — Causes generation of a listing of the modules in the library after all other requested operations have been performed. If the **-s** option is used on the command line which invokes *oml*, the listing will include the public symbols defined in each module as well as a list of the module names themselves.

@[filename]	Causes the remainder of command input to be read from **stdin**, or from the named file. Note: brackets are used to show that the file name is optional, and are not included if a file name is specified).

If replacement modules or deletions are specified, a new version of the library file will be built, provided that no errors are detected. This new version is created first as a temporary file; when it has been completely built, the original library file (if it existed) is deleted, and the temporary file renamed. This sequence insures that the original library file will not be affected if an error is detected. Modules are always included in a library in topologically sorted order, so that no backward references occur (except in the case of modules which reference each other, which are retained in the same order in which they are encountered).

Warning messages are produced if:

- A module named in a deletion or extraction list was not found in the library.
- If a second definition for a public symbol is encountered in one of the modules to be included.

SEE

blink,lc

EXAMPLE

The following examples illustrate the use of *oml*. Remember that replacing modules within a library file only works correctly if the module name is the same as the file name. Since a single object file is essentially a library of one module, *oml* can be used to find out what module name is included in the object file by using the following command:

```
oml name.o l
```

Building a New Library	Create a list of the file names of the object modules which will make up the library. The Lattice **extract**

utility can be used to accomplish this. Then create the library using the following command:

```
oml new.lib r @name.lst
```

where **new.lib** is the name of the library to be created, and **name.lst** contains a list of the files to be included in the library. Note that creation of a new library is one occasion where the correspondence between file and module names is not required.

Extracting Modules from a Library

Use the following command to break out all of the modules from a library:

```
oml -o:object/ cfuncs.lib x *
```

Note that this command will be successful only if no module names in the library CFUNCS.L contain path names. A file for each of the modules in the library will be created in the directory **:object/** in this example.

Deleting Modules from a Library

Use the following command to delete modules from a library:

```
oml cfuncs.lib d tribe.o
```

This example deletes the module **tribe.o** from the library **cfuncs.lib**.

Listing the Modules in a Library

Use the following command to obtain a listing of the modules and symbols in a library file:

```
oml -s test.lib l
```

The listing may be saved to a file using I/O redirection:

```
oml >test.lst -s test.lib l
```

NAME

splat	Stream editor for performing character substitution

SYNOPSIS

```
splat [-s] [-o | dPREFIX] [-v] pattern string file1 fileN
```

DESCRIPTION

splat is a stream editor for substituting character patterns on individual or groups of files. The original version of the file is left unchanged unless specified. This feature in conjunction with the stream operation of **splat** means a very efficient way of handling character substitution.

The flags shown on the command line above are:

-dPREFIX This option will specify a directory prefix indicating the directory where the edited file(s) are to be placed.

-o This option will overwrite the original version of the file with the new version.

pattern This option will match a pattern as described in the section dealing with **GREP**.

-s This option will echo to the standard output device the name of the current file. In addition, when no character substitutions have been performed on the file, this information is echoed to the output device. Without this option, **splat** performs its task silently.

string This option will designate the string to be substituted for the pattern.

-v This option will display all lines in which substitutions are made to standard output. No files are created when using this option.

NAME

tb Traceback Utility

SYNOPSIS

```
tb [-<options>] [[tbfile] file]
```

DESCRIPTION

This command is used to process a **Snapshot.TB** file produced by a program linked with **catch.o**. This file is a standard IFF format file that contains information about the program abend and environment at the time of the exception.

If given, *options* must be a minus sign followed by one or more of the following characters in any order.

l	This will dump all sections present in the traceback file.
x	Symbol Xref. This dumps the location of all symbols encountered in the program.
s	Stack. This dumps the contents of the entire stack at the time of the snapshot.
r	Registers. This displays the current contents of registers at the time of the snapshot.
v	Environment. This is the default to display where the program was executing and the call back chain.
m	Memory. If present in the snapshot file, this will display the amount of memory available at the time of the snapshot.
u	User data. If present in the snapshot file, this will display any program generated user data section.

If no options are given, **tb** will default to printing out the current stack frame call back to indicate the current code location.

The optional parameter **file** indicates a program to read the debugging information from. If not given, the **tb** command will use the program specified in the snapshot file. Note that this option is most useful when you create two images: one with full debugging information and another without debugging information that you normally use. In this way, you get the benefits of faster loading of the nondebug version yet still have access to the traceback information if it does crash.

The **tbfile** parameter is used to override the default of **Snaptshot.TB** in the current directory. If you give this option, you MUST also specify a program name.

SEE

lc, lc1

EXAMPLE

With a program **test** linked with **catch.o** that has created a file **Snapshot.TB** in the current directory. We run the **tb** command to produce a traceback:

```
tb
Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110  211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde =  foo line 5
   called from 22156a = main line 11
   called from 221c60 =  main + 704
   called from 2210ca = hunk 0 + $152
```

To obtain a full dump of all information, we can use the **-l** option:

```
tb
TraceDump 0.88 Copyright (c) 1988 The Software Distillery

TraceDump Utility: catch.o; Version 1, Revision 0
Processor type: 68000
VBlankFreq 60, PowerSupFreq 60

Symbols for hunk 0
                 foo = 22153c                main = 22155c

Program Name: testit; run from CLI
Program load map (addresses are APTRs, sizes are in bytes)
220f78 $1110  211268 $5b4
Terminated with GURU number 00000005, Divide by Zero Error
Error occurred at address 221cde = _foo line 5
   called from 22156a = main line 11
   called from 221c60 = _main + 704
   called from 2210ca = hunk 0 + $152
```

```
Registers:
D0=00221cde D1=00000000 D2=00000001 D3=00002718
D4=00000001 D5=0000002b D6=0000003b D7=00000000
A0=00221f54 A1=00243fcc A2=0024460a A3=00225c68
A4=00211268 A5=002445be A6=002033c8 A7=002445ae
PC 221cde   C=0 V=0 Z=1 N=0 X=0

stack top: 244640, stack pointer: 2445ae, stack length: $94
entire stack, size = $94 bytes
2445AE: 000003ED 00221558 0000002B 00000000 : ...m.".X...+....
2445BE: 002445CA 0022156A 00225C68 002445F6 : .$EJ.".j."\h.$E.
2445CE: 00221C60 00000001 00211620 00225CC4 : .".`.....!. ."\D
2445DE: 00226030 00225C68 83D10000 00000021 : ."`0."\h.Q.....!
2445EE: 13230021 00211620 00244640 002210CA : .#.!.!. .$F@.".J
2445FE: 00244602 74657374 6974000A 00000022 : .$F.testit....."
24460E: 60300000 27100000 27180000 00010000 : `0..'...'.......
24461E: 002B0000 003B0022 60300022 61300020 : .+...;."`0."a0.
24462E: 35300022 0F740024 464400FF 425800FF : 50.".t.$FD..BX..
24463E: 424C0022                            : BL."
```

The **tb** utility can present each individual piece of the information based on additional option letters. In the above dump, you can find the following sections:

(1) Xref section **-x** option

(2) Environment section **-e** option

(3) Register section **-r** option

(4) Stack section **-s** option

NAME

touch — Adjust the time stamp on specified files to the system time

SYNOPSIS

```
touch [-m] file1 fileN
```

DESCRIPTION

touch amends the time stamp on a file to the existing system time. This utility supports the use of wildcards - the **-m** option specifies MS-DOS convention. The default for wildcards is the AmigaDOS conventions of **#?** characters.

> Note that it is essential that the system date and time are accurate before attempting to use this utility. Failing to adhere to this caveat may cause unpredictable results.

NAME

wc	Tabulate the number of characters, words, lines within file

SYNOPSIS

```
wc [-p] [-s] file
```

DESCRIPTION

wc is a utility which is used to obtain statistical information about the contents of a file. The details of the file are directed to the standard output device - the screen.

The flags shown on the command line above are:

p	This option will filter the file to remove non-printable characters.
-s file	This option will calculate a checksum for the specified file.

The checksum calculation format consists of two parts - the number of printable ASCII characters and the checksum for the file. An example of the calculation format looks like this:

```
Characters  : 12391
  Words     : 1975
  Lines     : 130

Sum : 1919 0
```

Section 3

Lattice Amiga Environment Variables

This section describes the AmigaDOS environment variables that are used by the Lattice C Compiler and related programs.

NAME

include: Include search path

SYNOPSIS

```
assign INCLUDE: include path
```

DESCRIPTION

This logical name is used by the compiler to find files that are included via the **#include** statement. The **include path** should be the disk and directory specification that describe the location of the header files. For example,

```
assign INCLUDE: "Lattice C:include"
```

indicates that the compiler should look for included files in the directory **include** on the disk whose volume name is **Lattice C**. Note that if any portion of the path contains spaces, then the entire path must be enclosed in double quotes.

The compiler accepts two forms of the **#include** statement, as follows:

```
#include <file>

#include "file"
```

When the first form is used, the compiler looks for **file** only in the directory associated with the **INCLUDE:** logical name. With the second form, the compiler looks in the current directory, then in any directories specified via the **-i** option, and finally in the directory associated with the **INCLUDE:** logical name.

SEE

assign, lc, lc1

NAME

lc: Compiler executable file path

SYNOPSIS

```
assign LC: program path
```

DESCRIPTION

This is the logical name used by the driver program **lc** to locate the various programs required to compile and link source modules. This can be very useful on floppy disk based systems where there may not be sufficient online storage capacity to contain the compiler files in addition to other working files. By assigning **LC:** to the disk volume and directory containing the compiler and linker, **lc** will be able to locate and load the appropriate programs even if that disk volume isn't currently mounted. If required, you will be prompted to insert the necessary disk volume at the correct time.

For example,

```
assign LC: "Lattice C:"
```

will cause **lc** to load the compiler executable programs from the root level of the disk volume named **Lattice C**.

If the assignment is not made, **lc** looks for the programs in the default directory **lc/** in the root level of the system drive.

SEE

assign, lc

NAME

lib: Library file path

SYNOPSIS

```
assign LIB: library path
```

DESCRIPTION

This is the logical name used by **lc** to locate the Lattice C startup routine **c.o** and the Lattice and Amiga libraries for the linker. If the assignment has been made, the driver program, **lc**, will specify that the linker look in the directory associated with the logical name for the startup code and the standard libraries. The directory may be on any disk volume. If the volume is not mounted, you will be prompted to insert the correct volume at the appropriate time. For example,

```
assign LIB: "Lattice C:lib"
```

specifies directory **lib/** on disk volume **Lattice C** as the location of the libraries and startup code.

If the assignment is not made, the linker will be directed to look in the subdirectory **lib/** of the path used to locate the linker. In other words, if the logical name **LC:** is assigned, the libraries will be assumed to be in the directory **LC:lib**. If **LC:** is not assigned, the directory **SYS:lc/lib** will be used.

SEE

assign, lc, lc1

NAME

quad: Intermediate file path

SYNOPSIS

```
assign QUAD: quadfile path
```

DESCRIPTION

This is the logical name used to direct the compiler to place the intermediate file in some location other than the source file directory. By default, the compiler creates the intermediate, or quad, file in the same directory as the source file. You can use the **-q** option of the **lc** driver program or the **-o** option of lc1 to cause this file to be created somewhere else.

The **QUAD:** logical name is used by the driver program to assign a different default location for the intermediate file. For example,

```
assign QUAD: ram:
```

will cause the compiler to use ram disk for the intermediate file when it is invoked via the **lc** command.

SEE

assign, lc, lc1

Appendix A

Compiler Command Summary

The following table summarizes all of the options for the Lattice Amiga C Compiler **lc**, **lc1**, and **lc2** commands.

LC	LC1	LC2	Meaning
-ab		-cb	Chip BSS
-ac		-cc	Chip Code
-ad		-cd	Chip Data
-b	-b		Base relative data
-b0	-b0		Non-Base relative data
-b1	-b1		Base Relative Data
-c+	-c+		Suppress structure messages
-ca	-ca		ANSI compatibility
-cc	-cc		Allow nested comments
-cd	-cd		allow $ in identifiers
-ce	-ce		Suppress error line printing
-cf	-cf		Require function prototypes
-ci	-ci		Suppress multiple includes of same file
-ck	-ck		Allow new keywords
-cm	-cm		Allow multiple character constants
-co	-co		Enable old style preprocessor
-cr	-cr		Allow register parameter parsing
-cs	-cs		Create only one copy of identical strings
-ct	-ct		Enable warnings for tags used without definition
-cu	-cu		Force all char declarations as unsigned char
-cw	-cw		Shut off warning for return without a return
-cx	-cx		Treat all global declarations as externals
-C			Continue on error
-d	-d		Enable debugging
-d0	-d0		Disable debugging
-d1	-d1		Enable debugging - dump line table
-d2	-d2		Generate symbol information
-d3	-d3		Generate symbol info, dump at every line
-d4	-d4		Generate full symbol information
-d5	-d5		Generate full symbol info, dump at line
-dx=y	-dx=y		Define preprocessor symbol
-f	-f		Use Motorola FFP
-ff	-ff		Use Motorola FFP
-f8	-f8		Generate code for Motorola 68881
-fi	-fi		Generate code for IEEE libraries
-fl	-fl		Use standard Lattice libraries
-gd	-gd		Cross reference defined symbols
-gc	-gc		Cross reference compiler provided files
-ge	-ge		List excluded lines
-gh	-gh		List header files
-gi	-gi		list included files
-gm	-gm		List macro expansions
-gn	-gn		Print narrow lines
-gs	-gs		List source
-gx	-gx		Produce cross reference listing
-Hxxx	-hxxx		Read in header file xxx
-hb		-hb	Fast BSS
-hc		-hc	Fast Code
-hd		-hd	Fast Data
-ix	-ix		Specify include directory
-j<n>	-j<n>		Disable message #n

LC	LC1	LC2	Meaning
-j<n>e	-j<n>e		Make message #n an error instead of a warning
-j<n>i	-j<n>i		Disable message #n
-j<n>w	-j<n>w		Enable message #n as a warning
-L+			Specify additional linker objects
-La			Addsym
-Lc			small code memory model
-Ld			small data memory model
-Lf			ffp math library
-Lh			hunk map option
-Ll			library map option
-Lm			math library
-Ln			nodebug option
-Lo			overlay map option
-Ls			symbol map option
-Lt			Tiny modules
-Lv			verbose option
-Lx			x-ref map option
-l	-l		Align objects on longword boundaries
-M			only compile modified source files
-m		-m	Generate code for Motorola 68000
-m0		-m0	Generate code for Motorola 68000
-m1		-m1	Generate code for Motorola 68010
-m2		-m2	Generate code for Motorola 68020
-m3		-m3	Generate code for Motorola 68030
-ma		-ma	Generate code for All Motorola processors
-mc		-mc	Disable cleanup overhead reduction enhancement
-mr		-mr	Disable automatic registerization
-ms		-ms	Generate code optimized for space
-mt		-mt	Generate code optimized for time
-n	-n		Retain only 8 characters for identifiers
-O			Invoke Global Optimizer
-ox		-ox	Place object file in location x
-p		-p	Generate stack probe instructions
-p	-p		Preprocess only
-pe	-pe		Generate prototypes only for externs
-ph	-ph		Generated precompiled header file
-pp	-pp		Generate prototypes with _PARMS for portability
-pr	-pr		Generate prototype file
-ps	-ps		Generate prototypes only for static functions
-qx	-ox		Place quad file in location x
-q<n>e	-q<n>e		Quit compilation after n error/warnings
-q<n>w	-q<n>w		Quit compilation after n warnings
-q			Same as -q1e1w
-r	-r		Default all subroutine calls to NEAR (PC-relative)
-r0	-r0		Default all subroutine calls to FAR (Absolute)
-r1	-r1		Default all subroutine calls to NEAR (PC-Relative)
-rr	-rr		Default all subroutine calls/entries to Register conventions
-rs	-rs		Default all subroutine calls/entries to Standard/Stack conventions
-rb	-rb		Generate code for both Register and stack convention entries
-Rx			Place compiled objects into library x
-s		-s	Specify default segment names
-sb=x		-sb=x	Specify name for BSS segment
-sc=x		-sc=x	Specify name for code segment
-sd=x		-sd=x	Specify name for data segment

LC	LC1	LC2	Meaning
-u	-u		Undefine all preprocessor symbols
-ux	-ux		Undefine preprocessor symbol x
-v		-v	Disable stack checking code
-w	-w		Default to short integers
-x	-x		Treat all global declarations as externals
-y		-y	Load up A4 with base address at start

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

Q

R

S

T

W

Editor

Editor

Editor

Lattice® Screen Editor (LSE ™)

Reference Manual

Version 2.0

Part of the Lattice C Compiler for AmigaDOS

Lattice, Incorporated
2500 S. Highland Avenue
Lombard, IL 60148
USA

Subsidiary of SAS Institute Inc.

Lattice Screen Editor (LSE) Reference Manual

This document was produced using Lattice HighStyle™

Lattice® Screen Editor (LSE™)

Reference Manual

Version 2.0

Part of the SAS/C® Compiler for AmigaDOS™

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
USA

Lattice® Screen Editor (LSE™) Reference Manual

This document was produced using ***HighStyle***®, the Lattice Document Composition System.

Table of Contents

Section 1

Overview of LSE

1.1 Introduction

This major part of the Lattice Amiga C Compiler Manual is the *Lattice Screen Editor Reference Manual.* It describes the actions, operations and features of version 2.0 of the **Lattice Screen Editor**.

1.1.1 Features of LSE

The **Lattice Screen Editor** (**LSE™**) is an editor that is easy to use and it offers the features, flexibility and power needed by professional programmers. The LSE editor includes:

- Amiga mouse pull-down menus, click on gadgets, click on text, and scrollbar,
- Programmable keystrokes to tailor LSE to your personal preferences,
- Integrated Compile and Edit capabilities,
- Nested UNDO capabilities of up to 50 levels,
- The ability to edit up to nine files at the same time,

- Easy to use mouse and menu commands,
- Automatic file back-up,
- Special input modes for assembly language, automatic tabbing, word wrap for text mode, and auto-indenting on opening brace ({),
- Pattern and string searches,
- Keystroke saver macros, and
- On-line Help.

1.1.2 Contents

The *Lattice Screen Editor Reference* is divided into the following six sections and four appendices:

- **Section 1** presents an overview of **LSE** and contains a list of the **LSE** features.
- **Section 2** covers the **LSE** environment, loading the system, and the command line options.
- **Section 3** describes the basics of editing in **LSE** and covers the screen, entering text and commands, and the data, message, and help files.
- **Section 4** describes the Special Keys which control editing, and entering text into a file.
- **Section 5** describes how to compile Lattice C source files from the **LSE** integrated environment.
- **Section 6** describes the Mode Menu options which control the editing environment.
- **Appendix A** presents a list of the error messages which may be generated during execution.
- **Appendix B** describes installing and customizing **LSE**.
- **Appendix C** reviews the special keys used while entering text.
- **Appendix D** describes the regular expression syntax accepted by **LSE**.

1.2 This Section

Section 1 presents an overview of the **Lattice Screen Editor (LSE)**.

1.3 An Editor for Programmers

LSE is written in Lattice C and has been specifically optimized to run on the Amiga.

1.3.1 Built to Reduce Your Development Time

Now you can use **LSE** to write C source code and compile code directly from memory. Pressing a single key invokes the Lattice C Compiler using your predefined compiler and error checking options.

Using this integrated compile/edit environment you can compile the source file while buffering compiler syntax errors for later review. Prompts are displayed only during the review cycle.

You can also elect to check the compiler output either for errors only or for both errors and warnings.

This integrated concept will greatly reduce your compile/edit cycle which will increase your programming productivity.

1.3.2 Up to Speed Quickly

You don't want to waste time learning a new editor, but you want the ability to compile from memory. So in addition to **LSE**'s standard command structure, we've included a configuration program that enables you to tailor **LSE** to your personal keystroke preferences.

- **LSE** uses spreadsheet-style menus and pulled-down menus with the Amiga mouse and standard Amiga gadgets to get you going fast. And as you become more familiar with the editor you can execute commands via shortcut keys without using the menus.
- Using the **LSE** Installation Program you can tailor **LSE** keystrokes to

match those of your current editor. And you can even change the **LSE** messages or prompts by editing the **LSE** message file.

1.3.3 Editing Modes

LSE includes special language specific input modes for C language and assembly language programmers as well as a power-type (word-wrap) mode. **LSE** can handle files up to the limit of memory on your equipment so you don't have to worry about file sizes.

1.3.4 Nine Things at Once

Like any good programmer you are concerned about consistency and continuity. You want to check out a module you coded at the start of the project to make sure that your current work is structured the same way. And you want to pull out a few lines of code from there and copy them here.

That's why **LSE** can do nine things at once. You can keep up to nine files in memory, viewing up to two files at the same time. And under AmigaDOS you can fork **LSE** leaving it in the background in order to start a compile or perform any other Amiga system task, then switch back and continue editing.

1.4 LSE Features

Besides the standard functions you expect in an editor (block manipulation, full screen data entry, etc.,) **LSE** provides you with:

Lattice C Compiling and Error Tracking — C files can be compiled from memory and the output from the compiler can be processed by **LSE** to find lines in your source code containing errors or warnings. This fast compile/edit loop can greatly reduce your program development time.

Multi-Window Environment — **LSE** allows you to keep nine files in memory. You can display two files at the same time and

move and copy data between the files quickly and easily. In addition, a toggle is provided for switching between a full screen display and a split screen display.

Multi-Level Commands

All **LSE** commands are available either through spreadsheet-style menus or short-cut keys. A mouse can be used to select menu items or to move the editing cursor. Using the mouse you can pull-down the top menus, you can click on gadgets, you can click on text positioning of the cursor, and you can use the scrollbar.

Backup Directory

You can target back-up files either to the current directory (with an extension of *.bak*) or to a default directory. The automatic back-up feature can be disabled.

Special Input Modes

LSE supports three assembly language input modes, giving case folding and the designation of a comment character. A power-type (word-wrap) mode is also available for preparing text to be further manipulated by other text processing utilities.

Full 8-Bit Character Input

LSE supports the entry of any 8-bit character into the text. This allows **LSE** to support special graphics characters and foreign character sets.

Installation Program

An installation program is provided so that you can customize any or all of **LSE**'s keystrokes. You may change menus, prompts, and help messages by editing the message files directly.

Pattern and String Searches

Text searching can use patterns as well as simple strings. Patterns are formed as *regular expressions* as in the popular *grep* utility.

Keysaver Macros

Macros can be created using **LSE**'s Keysaver Macro options. Macros can be any combination of commands and characters strings. Macros can be saved for later use or eliminated after an editing session. A two key combination can be designated to start the playback of any saved Macro character string.

Auto-Indenting on Open Brace ({)

To make C programming easier, you can auto-indent on each open brace ({) character. This is a toggle option which can be switched off when using **LSE** for other uses.

On-Line Help

F1 provides on-screen help at virtually any level of **LSE** processing.

Section 2

Getting Started

2.1 Introduction

Section 2 describes getting started with **LSE**. It covers the **LSE** environment, loading the system, and the command line options.

2.2 The LSE Environment

LSE requires an Amiga system with at least 512K bytes of application memory. A hard drive is preferred but **LSE** operates well on a system that has only floppy drives.

2.3 Installing LSE

LSE is included as part of the *Lattice C Compiler*. Refer to the *Lattice C Compiler User's Guide* for complete information on installing **LSE** on your system.

2.4 The Command Line

The command line to invoke **LSE** is:

```
lse [-v | [[-ddatafile] file1[.ext] [file2[.ext]]]
```

where optional information is shown enclosed in brackets ([ ]) and where:

-v — This option displays the **LSE** version number and copyright. You will need the version number from this display should you call for support. The screen is cleared and your file is displayed in an **LSE** window.

-ddatafile — The option *-d* indicates that a datafile name is being entered. The datafile is the name of the file created by the installation program **LSEINST.EXE**. The default is to search the directories in the PATH variable for **LSE.DAT**. There must be no blank characters between the option *-d* and the datafile name.

file1[.ext] — The file to be read into window 1. The extension is optional. Using **LSEINST** you can choose to automatically append default extensions to the file name (like **.c** for C source files).

file2[.ext] — The file to be optionally read into window 2. Again, the extension is optional and appended only if the default (no extension) has been changed.

You can invoke **LSE** without a filename by entering:

```
lse ↵
```

Use the *Project Menu* and the *Rename Option* to name a text file invoked without a filename. **LSE** will not allow you to save an un-named text file.

Section 3

Using LSE

3.1 Introduction

Section 3 describes the basics of running **LSE** and covers the screen, entering text and commands, and the data, message and help files.

3.2 The LSE Screen

You edit your text file in a *window*. A window of text refers to the area of the screen reserved for text. The default text window size is 20 lines by 78 columns. Up to nine files can be open at any one time. The files can be displayed in any open window as nine full screen displays to toggle between; or a split screen display showing two open windows at any one time.

3.2.1 The Current Window

The *current window* is either the full window being displayed, as indicated by the current active window indicator, or the window which contains the cursor when two windows are displayed in split screen.

A window in full screen uses all but the last three lines of the screen area. The last three lines of the screen area contain the *status line*, the *prompt line*, and the *input/descriptor line*.

3.2.2 The Status Line

The *status line* contains the following information indicators:

- Current line
- Current column
- Current filename
- Current mode indicator
- Window number
- Keystroke saver active indicator
- Marked block indicator

The *status line*, with all indicators displayed, looks like this:

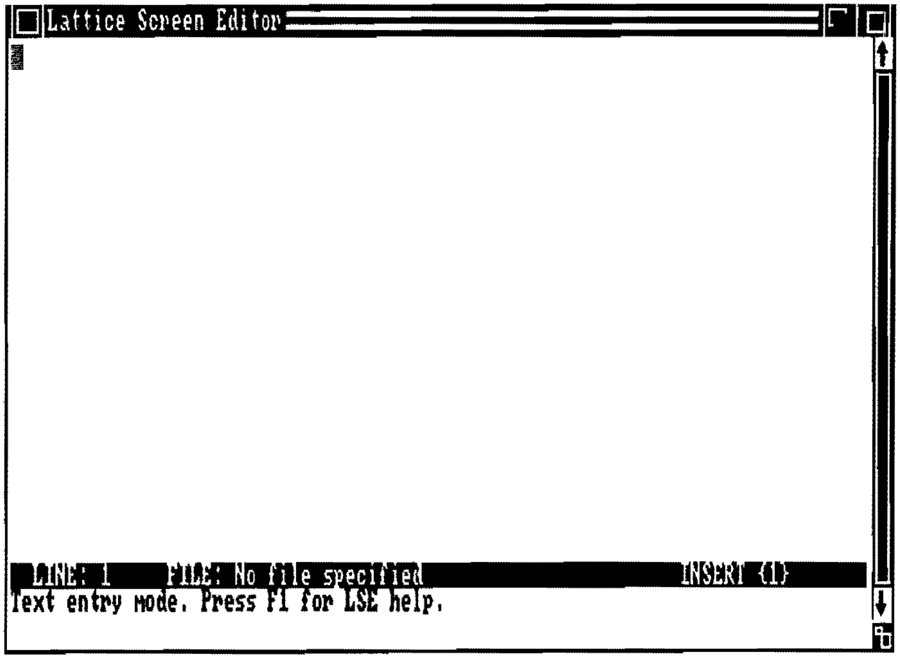


Figure 3.1 The Status Line for a Single Open Window

When more than one window is displayed at a time each status line refers to the window directly above it:

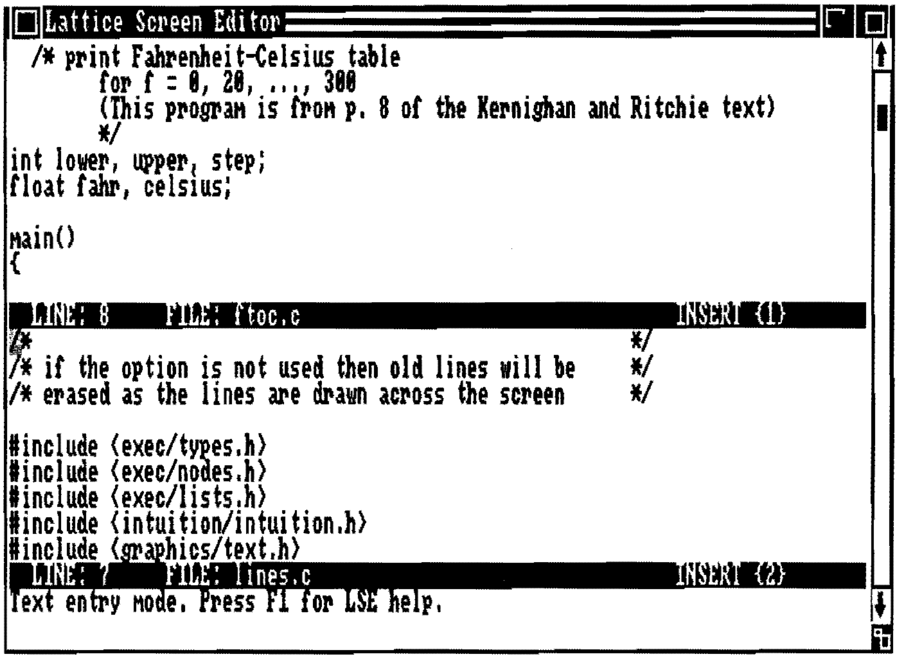


Figure 3.2 The Status Lines for Two Open Windows

3.2.2.1 Current Line Indicator

The *current line* is that line of text on which the cursor resides. The current line indicator displays the line number of the current line:

```
LINE: 80
```

This indicates that the current line of the file is line 80.

3.2.2.2 Current Column Indicator

The *current column* display is optional and is activated either from the *Mode*

Menu during an editing session or from the installation program (to change the default). The default is **OFF**. When **ON**, the column indicator displays the column at which the cursor is positioned:

```
COL 6
```

This indicates that the cursor is positioned at column 6.

3.2.2.3 Current Filename Indicator

The *current filename* is the name of the file from which the text was read. It is also the name of the file that the text will be written to when the *Project Menu/Save Option* has been selected:

```
FILE: ftoc.c
```

The above example indicates that *ftoc.c* is currently begin edited. Note that the current filename is absolute even if a relative filename was entered.

3.2.2.4 Current Mode Indicator

The *current mode* indicator, displays either *INSERT* or *OVERWRITE* to indicate which input mode you are in.

3.2.2.5 Window Number Indicator

The *window number* indicates which window is displayed. The indicator displays the number 1 through 9, between braces, to indicate which window you are editing. When editing window 3:

```
{3}
```

is displayed on the *status line*.

3.2.2.6 Keystroke Saver Active Indicator

The *keystroke saver active indicator* is displayed only when you have invoked the macro keystroke saver. When active, the number of the active macro (from 1 through 10) is displayed. All keystrokes will be recorded and stored in that macro.

3.2.2.7 Marked Block Indicator

There are three components to the *marked block* indicator:

- The left bracket character ([) is displayed if a Beginning of Block has been set.
- If an End of Block has been set, the right bracket character (]) is displayed.
- If the block is marked successfully, the character * is displayed between the brackets.

After a block has been marked successfully:

```
[*]
```

is displayed on the *status line*.

3.2.2.8 Error Messages

The *status line* is also used to display error messages. Once an error message has been displayed, it remains on the *status line* until the cursor is moved off the current line of the text file. Once the cursor is moved off the current line, the *status line* indicators are again displayed. Error messages are described in *Appendix A*.

3.2.3 The Prompt Line

There is one *prompt line* for **LSE** regardless of the number of open windows.

The *prompt line* is located directly below the *status line* (the window 2 *status line* when more than one window is displayed).

The *prompt line* displays prompts and messages such as menus, requests for filenames, and verification of a string replace.

3.2.4 The Input Line

The *input line* is displayed one line below the *prompt line* on the last line of the **LSE** screen. Your responses to the *prompt line* are displayed on the *input line* including input for filenames, strings, and line numbers.

The *input line* is also used to display messages that **LSE** is performing such tasks as **Opening...**, **Closing...**, or **Compiling...**.

The input line also acts as the *descriptor line* when invoking **LSE** menus that identify an action to be performed.

3.3 Entering Text and Commands

When invoking **LSE** you may use one or two filenames along with the command, **lse**. If more than one filename is specified, the first file on the command line is read into the top window and the second file on the command line is read into the bottom window.

An empty window is displayed if a file did not previously exist or contained no text. If you invoke **LSE** without a filename and want to save the file after editing, use the *Project Menu/Rename Option* to name the file before saving it. Refer to *Section 2* for information on the command line.

When you access **LSE** you are in the *Insert Mode*. This means that entering any one of the 96 printable ASCII characters (hex 20 through hex 7f) inserts that character into the text at the current character position. The Insert Mode is the default. You can toggle into the Overwrite Mode by pressing Ins (or the 0 on an Amiga 1000).

When you are in the Overwrite Mode, entering a character overwrites the character at the current cursor position. Any other 8-bit character code can be entered into the text by preceding it with the *escape character*, as described in *Section 4*.

All actions in **LSE** start at the current line. When entering characters causes the width of the current line to exceed the current window width, the current line is re-framed. As the column of the current character changes, the current line is re-framed to keep the current character position in the window at all times. As soon as the cursor is moved off the long line, that line is re-framed in the window starting at column 1. The maximum width for any line in **LSE** is 256 characters.

Any 8-bit character code not in the 96 printable ASCII character set and input without being preceded by the escape character is interpreted by **LSE** as an attempt to execute a command. If the character code does not map to one of the over forty **LSE** functions, the code is entered into the text. The character will be displayed in reverse video.

If the character code matches one of the special keys, **LSE** performs the function, possibly repositioning the cursor, and passes control back to you for further text entry.

If the character code matches an **LSE** command you are prompted for more information or a menu is displayed.

3.4 Data, Message, and Help Files

The following files control all actions of **LSE**:

- The data file, **LSE.DAT**, contains information about your computer and your key assignments.
- The message file, **LSE.MSG**, contains text for the menus, help messages, error messages, and prompts.
- The help file, **LSE.HLP**, contains all help screen text.

LSE.DAT and **LSE.MSG** are read when **LSE** is invoked. **LSE** searches for **LSE.HLP** when on-screen help is requested.

You can change **LSE** by altering any or all of these files:

- **LSE.DAT** is changed using **LSEINST**, the installation program.
- **LSE.MSG** and **LSE.HLP** are both text files and can be changed using **LSE**.

See *Appendix B* for more information before making any changes to these files.

Section 4

Special Keys

4.1 Introduction

Section 4 describes the Special Keys that control **LSE**. The Special Keys control editing and entering text into a file. Special Keys also trigger the Commands which control the processing performed while in an editing session.

There are eleven categories of Special Keys:

Mouse Controls The Amiga mouse is given special control to: move the cursor position within the file by clicking on text, pull-down menus from the top that allow you to have all the LSE command access from the mouse, move through the text via scroll gadgets, and click on standard gadgets. LSE uses all of the standard Amiga system gadgets.

Cursor Movement The Cursor Movement keys allow you to position the cursor within the file: up or down a line; to the previous or next character or word; and to the beginning or end of the line. They also allow you to display the previous and next page of text, the beginning and end of text, and scroll the display up and down.

Change Window The Change Window keys allow you to change the active window; change the window size; and cycle through hidden windows.

Insert The Insert keys allow you to toggle between the Insert Mode and the Overwrite Mode; insert a blank line; and restore lines that have been previously deleted.

Delete The Delete keys allow you to remove text from the file by character, word, line and to the end of the line.

Help The Help key allows you to display context-sensitive help windows.

Change Color The Change Color keys allow you to change the foreground or background color of the display screen.

Keysaver Macro The Keysaver Macro keys allow you to define and execute macros.

Mark Block The Mark Block keys allow you to mark the beginning and the end of a block of text that you want to copy, delete, move, print, read, or write.

Command The Command keys allow you to control the processing performed while in an editing session: performing block operations, invoking the Lattice C Compiler, forking the command processor, displaying the *Mode Menu*, displaying the *Project Menu*, opening a new window, moving to a specific line number, searching for a text string, replacing a text string, quitting and undoing a file action.

Action The Action keys allow you to to control actions in conjunction with other options or commands: displaying the *Main Menu*; displaying the next error; entering the escape character to the file; locating the next match for a search; switching to Interlace mode; and setting compiler options.

The standard keystroke assignments may be changed or customized using **LSEINST** as described in *Appendix B*.

4.2 Mouse Controls

The Amiga mouse controls for LSE allow for the following capabilities:

Mouse Controls

Action	Function
Button Left Click on text	Move cursor to click point
Button Left Click on slider-bar	Move to relative text position
Button Right Click on pull-down menu	Execute LSE command being pointed at
Button Left Click on gadget	Execute standard gadget command being pointed at

4.3 Cursor Movement Keys

These keys reposition the cursor within the file:

Function	Key	Action
Up One Line	↑	Move the cursor up one line.
Down One Line	↓	Move the cursor down one line.
Previous Character	←	Move the cursor one character left.
Previous Word	Ctrl + ←	Move the cursor one word left.
Next Character	→	Move the cursor one character right.
Next Word	Ctrl + →	Move the cursor one word right.
Beginning of Line	Home	Move the cursor to the first character of the current line.
Beginning of Text	Ctrl + Home	Move the cursor to the first character of text file.

End of Line	End	Move the cursor to the last character of the current line.
End of Text	Ctrl + End	Move the cursor to the last character of text file.
Previous Page	PgUp	Display the previous page of text.
Next Page	PgDn	Display the next page of text.
Scroll Up	Ctrl + ^	Scrolls the display up without moving the cursor.
Scroll Down	Ctrl + V	Scrolls the display down without moving the cursor.

4.4 Change Window Keys

The Change Window keys allow you to change the active window, change the window size, and cycle through hidden windows:

Function	Key	Action
Change Window	Ctrl + F6	Change active window when in the split screen mode.
Change Window Size	F9	Toggle window display size from full to half screen.
Cycle Windows	F6	Cycle through hidden windows.

4.4.1 Change Window (Ctrl + F6)

The Ctrl + F6 key is a toggle that switches the cursor between text files in the split screen mode. The cursor is positioned at the same relative location in the new window as it was in the other window before you first pressed Ctrl + F6. If only one window is open when you press Ctrl + F6:

```
ERROR: NO ALTERNATE WINDOW AVAILABLE
```

is displayed on the *status line*.

4.4.2 Change Window Size (F9)

The F9 key causes the display size of the current window to be changed. If your current display of two open windows consists of a split screen, your new display will be full screen for each of the two windows. If your current display of two open windows consists of a full screen for each of the windows, your new display will be split screen.

If only one window is open when you press F9:

```
ERROR: ONE WINDOW OPEN — DISPLAY SIZE CANNOT BE CHANGED
```

is displayed on the *status line*.

4.4.3 Cycle Windows (F6)

The F6 key displays any text files loaded but not displayed. If only one window is available when you press F6:

```
ERROR: NO ALTERNATE WINDOW AVAILABLE
```

is displayed on the *status line*.

4.5 Insert Keys

The Insert keys allow you to toggle between the Insert Mode and the Overwrite Mode, insert a blank line, and restore lines that have been previously deleted:

Function	Key	Action
Insert Mode Toggle	Ins	Toggle between the Insert Mode and the Overwrite Mode.
Insert a Line	Ctrl + N	Insert a line before the current line, do not break the current line.
Restore Deleted Line	Alt + Y	Restore the last line deleted with Ctrl + Y at the cursor position, shifting the current line down one line.

4.6 Delete Keys

The Delete Keys remove text from the file by character, word, line and to the end of the line:

Function	Key	Action
Delete Current Character	Del	Delete current character from text.
Delete Previous Character	BkSp	Delete previous character from the text.
Delete Word	Ctrl + W	Delete text until the next word is encountered. A word is any contiguous text bounded by blanks.
Delete Line	Ctrl + Y	Delete the current line. The line can be restored using Alt + Y.
Delete to End of Line	Ctrl + E	Delete text from the cursor position to the end of the current line.

4.7 The Help Key

The F1 key (or HELP) is used to display context-sensitive help windows. Pressing F1:

- While entering text causes command key information to be displayed.
- While in the Command Menu causes menu information to be displayed.
- While in an individual option causes information on that option to be displayed.

4.8 Change Color Keys

The Change Color keys cycle the monitor foreground and background colors. You must have a color monitor for these keys to be effective:

Function	**Key**	**Action**
Cycle Foreground	F7	Cycle through the foreground colors.
Cycle Background	F8	Cycle through the background colors.

The color settings you have selected when you leave **LSE** become the default colors the next time **LSE** is started. You may not set the foreground and background to the same color.

4.9 Keysaver Macro Keys

The Keysaver Macro keys let you define and execute macros:

Function	**Key**	**Action**
Start/Stop Keysaver Macro	Alt + #	Start or stop recording keys to the keysaver macro.
Replay a Macro	Alt + F#	Replay a macro.

Note that the "#" refers to any number on the keypad (1,2,3,...,9,0). The number "0" will refer to function key F10 when you replay the macro.

4.9.1 Start or Stop Keysaver Macro (Alt+#)

To start a Keysaver Macro press Alt+# (a number key from 1 to 0). The following is then displayed on the *prompt line*:

```
Key Saver Begun
```

and the number of the macro being defined will be shown on the right hand side of the *status line*. All keystrokes will now be saved to a buffer. To end the macro press Alt and the same number key again.

4.9.2 Replay a Macro (Alt+F#)

Pressing Alt+F# replays the macro with that number. Each keysaver corresponds to a function key (from F1 to F10). For example, pressing Alt+F2 invokes macro number 2. No action occurs if Alt+F# is pressed with a number of a keysaver containing no keystrokes.

4.9.3 Saving Macros

If you do not save macros, they are lost when you complete your editing session and terminate **LSE**.

Use the *Project Menu/Macros Option* to save the macros you have defined. Select *Save* from the Macros sub-menu. You will then be prompted for a filename in which to save the macros.

```
SAVE MACRO FILE: SPECIFY MACRO FILENAME (default is LSE.MAC)
```

Enter a valid filename and press the ↵. Press F2 to leave the Save Macros display without saving the macro file.

4.9.4 Loading a Macro File

Once you have saved your macros to a Macro File you can load that file using the Load Macro option.

Use the *Project Menu/Macros Option* to load a macro file previously defined. Select *Load* from the Macros sub-menu. You will then be prompted for the macro file you want to use:

```
LOAD MACRO FILE: SPECIFY MACRO FILENAME (default is LSE.MAC)
```

Enter a filename and press the ↵. The file you have named will be written over the current (if any) macro file. There is no check to ensure that the named file is a valid macro file. Press F2 to leave the Load Macros display without loading a macro file.

4.10 Mark Block Keys

The Mark Block keys are used to mark the beginning and the end of a block of text that you want to copy, delete, move, print, read or write:

Function	Key	Action
Mark Beginning of Block	Ctrl + [	Mark the start of a block of text.
Mark End of Block	Ctrl +]	Mark the end of a block of text.

4.10.1 Mark Beginning of Block (Ctrl + [)

The Ctrl + [key identifies the start of a text block that will be used by the Block command. Ctrl + [is inclusive, which means that the designated block will include the current character.

When marking the beginning of a block of text, position the cursor over the first character of the block and press Ctrl + [. When the beginning mark is set, the character [is displayed on the *status line* of the window.

4.10.2 Mark End of Block (Ctrl +])

The Ctrl +] key identifies the end of a text block that will be used by the Block command. Ctrl +] is non-inclusive, which means that the designated block will not include the current character. When marking the end of a block of text, position the cursor to the right of the last character of the text block and press Ctrl +].

When the end mark is set, the character] is displayed on the *status line* of the window. When both start and the end of the block are marked, an asterisk is displayed between the two brackets (i.e., [*]) and the block is shown in reverse video. There may be only one active block marked at a time.

4.11 Command Keys

Commands allow you to control the processing within **LSE**. The Command keys are short-cuts which can be used in place of selections from the *Main Menu* displayed when F2 is pressed. The following table presents the function, the *Main Menu* selection, the short-cut keys and the description of the function from the *Main Menu*:

Function	Menu Select	Direct	Description
Block	B	Ctrl + B	Block Operations
Compile	C	F4	Invoke Lattice C Compiler
Fork	F	Ctrl + F	Fork Command Processor
Mode	M	Ctrl + M	Select LSE Mode
Line Number	L	Ctrl + L	Go to a Specific Line Number
Open	O	Ctrl + O	Open New Window

Project	P	Ctrl + P	Project Menu
Replace	R	Ctrl + R	Search and Replace
Search	S	Ctrl + S	Search
Quit	Q	F3	Quit
Undo	U	Ctrl + U	Undo Last Change

4.11.1 Block (Ctrl + B)

Pressing Ctrl + B or selecting *Block Operations* from the *Main Menu* causes the *Block Command Menu* to be displayed. Use the Block Command to perform any of the following operations:

- Copy a Block
- Delete a Block
- Move a Block
- Print a Block
- Read a Named File into the Current File Starting at the Cursor Position
- Write a Block to the Named File
- Move to the Beginning of a Block
- Move to the End of a Block

The *Copy Option* is highlighted when the *Block Command Menu* is displayed. To select an option from the menu:

- Position the highlighted bar over the selection using the ← and → keys, then select the option by pressing the ↵, or
- Enter the first character of the option and press the ↵.

4.11.2 Compile (F4)

Pressing F4 or selecting *Invoke Lattice C Compiler* from the *Main Menu* starts the compile of the edited C source file from memory. If the current file you are editing is not a C source file (it does not have an extension of .C, pressing F4 causes the message:

```
ERROR: TEXT IS NOT A .C FILE
```

to be displayed on the *status line*. See *Section 5* for more information on compiling from **LSE**.

4.11.3 Fork Command (Ctrl+F)

Fork allows you to leave your edited file and flip to the command processor in order to perform some system task. Pressing Ctrl+F or selecting *Fork Command Processor* from the *Main Menu* causes the message:

```
FORK COMMAND PROCESSOR (Press any key)
```

to be displayed on the *descriptor line*. Press any key to pop up a new window.

4.11.4 Mode Menu (Ctrl+M)

Pressing Ctrl+M or selecting *Select LSE Mode* from the *Main Menu* displays the *Mode Menu*.

The *Mode Menu* controls your input mode, your tab stops, expanding tab stops into spaces, automatic indenting on the { character, the column display on the *status line*, the type of search pattern utilized, whether an undo prompt is displayed, and error processing within the compiler. See *Section 6* for more information on the *Mode Menu*.

4.11.5 Go to Line Number (Ctrl+L)

Pressing Ctrl+L or selecting *Go to a Specific Line Number* displays the prompt:

```
ENTER LINE NUMBER TO LOCATE: Number
```

Enter a line number and press ↵. You are then positioned in your edited file at the specified line.

4.11.6 Open New Window (Ctrl + O)

LSE can open nine text files at the same time using the Open command. Pressing Ctrl+O or selecting *Open New Window* from the *Main Menu* causes the prompt:

```
OPEN WINDOW: Filename (Menu Key for view mode only)
```

to be displayed on the *descriptor line*. Enter a filename to open an additional window. To open a file in View Mode press F2 rather than entering a filename when this prompt is displayed. The message:

```
VIEW MODE FILENAME: filename
```

is then displayed. While files opened in View Mode cannot be changed, you are able to mark blocks of text for use by the *Block/Copy Option*; *Block/Print Option* and the *Block/Write Option*. No other actions can be performed on a file opened in View Mode.

If you have nine files opened and you press Ctrl+O, the message:

```
ERROR: NO WINDOW AVAILABLE
```

is displayed on the *status line* and no new window is opened.

4.11.7 Project Menu (Ctrl + P)

Pressing Ctrl+P or selecting *Project Menu* from the *Main Menu* causes the *Project Menu* to be displayed. Use the *Project Menu* to perform any of the following options:

- Quit Editing and do not Save File Changes.
- Quit Editing and Save File Changes.
- Save the File and Continue Editing the File.
- Save the File and Reopen Window for Next File.
- Do not Save the File and Reopen Window for Next File.
- Rename the Current File.

- Insert the Named File at the Current Line.
- Display the List of the Current Files.
- Load and Save Macro Files.

Figure 4.1 displays a screen with the *Project Menu*.

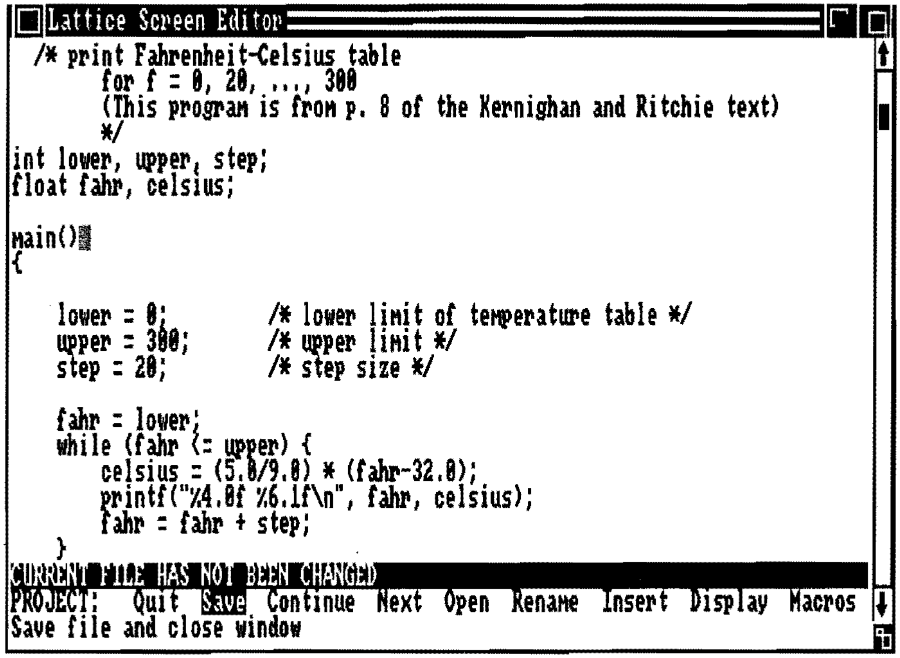


Figure 4.1 Display with the Project Menu

The *Save* option is highlighted when the *Project Menu* is displayed. To select an option from the menu:

- Use the ← and → keys to position the select bar over the command to be executed and press ↵.
- Press the letter corresponding to the selection you want.

4.11.7.1 Finishing an Editing Session

The *Project Menu* identifies the actions used by **LSE** to finish an editing session with the text file displayed in the current window.

LSE does not edit a file directly. When a file is named, it is copied to a work area while the actual (current) file remains unchanged. Whenever **LSE** saves text to a file, it then renames the current copy of that file to a backup copy and saves the edited text to the named file. By default, **LSE** renames the current copy to *filename.bak* erasing any previous version of *filename.bak*.

Using **LSEINST** you can optionally change the default file extension for the backup file as well as set up a backup directory. These options are explained in *Appendix B*.

4.11.7.2 File Status

Every time you access the *Project Menu*, **LSE** displays a file status message on the *status line* indicating if any changes have been made to the file:

- **CAUTION: CURRENT FILE HAS BEEN CHANGED** is displayed if the file has been changed.
- **CURRENT FILE HAS NOT BEEN CHANGED** is displayed if the file has not been changed.

4.11.7.3 Quit Option

The *Quit Option* abandons the text in the current window and then closes the current window:

- If the current window is the only active window, **LSE** is ended and control is passed back to the operating system.
- If there are two windows open, the non-active window becomes the current window. Once active, the new current window occupies all of the allotted screen space.

If you have made changes to the file and use the *Quit Option*, the file changes will be lost. Refer to the file status message if you are not sure.

> **NOTE:** It is faster to use the *close gadget* or Quit Command (F3) if you want to abandon the text files in two (or more) open windows rather than use the *Project Menu/Quit Option* for each open window. No file status message is displayed if text has not been changed. A warning prompt is displayed before files are abandoned if text has been changed in any window.

4.11.7.4 Save Option

The *Save Option* writes the text in the current window to the file specified by the current filename and then closes the window:

- If this is the only window open, **LSE** is ended and control is passed back to the operating system.
- If there are two (or more) windows open, the non-active window becomes the current window. Once active, the new current window occupies all of the allotted screen space.

You cannot save an unnamed file. If **LSE** was started without a filename and you have not used the *Project/Rename Option*, the message:

```
ERROR: NO FILE NAME SPECIFIED
```

is displayed on the *status line* and no further action is performed. Rename the file and use the *Save Option* again.

4.11.7.5 Continue Option

The *Continue Option* writes the text in the current window to the file specified by the current filename. The *Continue Option* does not close the current window or alter the current text in any way.

Like the *Save Option*, you must have named the file before using the *Continue Option*.

4.11.7.6 Next Option

The *Next Option* allows you to edit a series of text files without leaving **LSE**.

The *Next Option* writes the text in the current window to the file by the current filename. You are then prompted for the next filename to be loaded into the current window. Once the file has been loaded, the cursor is placed at the beginning of the text.

Like the *Save Option*, you must have named the file before using the *Next Option*.

4.11.7.7 Open Option

The *Open Option* abandons the text in the current window. You are then prompted for the next filename to be loaded into the current window. Once the file has been loaded, the cursor is positioned at the beginning of the text. Unlike the *Next Option*, the *Open Option* does not save the current text file before loading the new file.

If you have made changes to the file and use the Open option, the file changes will be lost. Refer to the file status message if you are not sure.

4.11.7.8 Rename Option

The *Rename Option* allows you to:

- Specify a new name for the file in the current window.
- Name a previously unnamed file when **LSE** is started without a filename.

An unnamed file cannot be written from memory using Save, Continue or Next. You must name the file using the *Rename Option* before using any of these other options. The current text is not changed when it is Renamed.

4.11.7.9 Insert Option

The *Insert Option* lets you insert a file on disk at the cursor position in your active window. It is functionally the same as the *Block Menu/Read Option*.

4.11.7.10 Display Option

The *Display Option* clears the editing window and presents the following display showing the active windows and the files open within those windows. You can press any key to return to your active window.

4.11.8 Replace ([Ctrl]+[R])

Use the *Replace Command* to locate text strings in your text file and replace them with the entered text string. Pressing [Ctrl]+[R] or selecting *Search and Replace* from the *Main Menu* causes the prompt:

```
REGULAR EXPRESSION SEARCH STRING: (MENU KEY to search back-
wards)
```

to be displayed. Enter the search string and press [↵]. The normal search and replace path is from the current cursor position "down" through the file to the end-of-file. To search and replace "up" from the current cursor position to the top-of-file press [F2] before entering your search string.

You will then be prompted for your replace string. Enter the replace string and press [↵]. The next prompt will ask whether you want to be prompted for each replace or if the replace should be performed through the file automatically.

You can change the search direction after your search and replace strings have been entered by re-displaying the search prompt, pressing [F2] and then [↵].

Use the *Mode Menu* or the install program to change from a regular expression search (as described in *Appendix D*) to a simple string search.

Use [Alt]+[S] to find the next occurrence of the entered search string. Once the

string is located, you will be prompted as to whether the replacement is to occur.

4.11.9 Search (Ctrl + S)

Use the *Search Command* to locate text strings within your text file. Pressing Ctrl + S or selecting *Search* from the *Main Menu* causes the prompt:

```
REGULAR EXPRESSION SEARCH STRING: (MENU KEY to search back-
wards)
```

to be displayed. Enter the search string and press ↵.

The normal search path is from the current cursor position "down" through the end-of-file. To search "up" from the current cursor position to the top-of-file press F2 before entering your search string. You can change the search direction after your search string has been entered by re-displaying the search prompt, pressing F2 and then ↵.

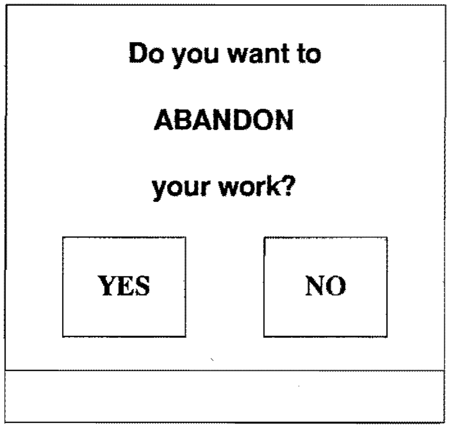
Use the *Mode Menu* or the install program to change from a regular expression search (as described in *Appendix D*) to a simple string search.

Use Alt + S to find the next occurrence of the entered search string.

4.11.10 Quit (F3)

Use the *Quit Command* to end an editing session without saving any file changes in any of the editing windows. Press F3 or select *Quit* from the *Main Menu* or the *close gadget* to quit an editing session.

Unlike the *Project Menu/Quit Option*, the *Quit Command* works on all windows rather than just the current window. No file status message is displayed if there have been no file changes. The warning requester:



is displayed if changes have occurred in any window. Select *No* to abort the *Quit Command*. Select *Yes* to abandon the changed file.

4.11.11 Undo (Ctrl + U)

The *Undo Command* allows you to recover from any file activity you have performed while still in the window for that file. Pressing Ctrl + U or selecting *Undo Last Change* from the *Main Menu* initiates the undo process. Using Undo, you can:

- Restore Replaced Text.
- Restore Deleted Block.
- Restore Deleted Line.
- Restore Replaced Line.
- Delete Inserted Line.
- Delete Added Block.

For example, if you used Ctrl + Y to delete a line of text from the file, pressing Ctrl + U causes the prompt:

```
UNDO — RESTORE REPLACED LINE?:  YES  NO
```

to be displayed on the *status line*. Enter *Y* to add the deleted line back to the file. Enter *N* to abort the Undo operation.

If you have not changed the default prompt before using the Undo Mode option, **LSE** will display a prompt (as shown above) before performing any action. If no prompt before Undo is displayed, the action is performed immediately when you select the Undo Command.

If you select the Undo Command when either no file actions have been performed on the text file in this window or all actions have already been undone, the message:

```
UNDO BUFFER EMPTY
```

is displayed on the *status line* and no action is performed.

Up to 50 actions are contained in the Undo buffer. An action is defined as any file activity which altered a contiguous part of the file from a single line to a defined block.

The levels are stacked and undone in LIFO (last-in/first-out) order. Separate Undo buffers are kept for each window. The Undo buffer for a window is emptied once you quit or save the file.

4.12 Action Keys

The Action Keys control actions in conjunction with other options or commands:

Function	Key	Action
Menu Toggle	F2	Display *Main Menu* or cancel menu item or action and return to editing.
Display Next Error	F5	Frame next compiler error.

Enter Escape Character	Ctrl + Esc	Enter escape character.
Find Next Match	Alt + S	Locate next match for search.
Interlace Toggle	ENTER	Interlace Toggle key.
Set Compiler Options	Ctrl + F4	Set compiler options.

4.12.1 Menu Toggle (F2)

Press F2 while editing a file to display the *Main Menu*:

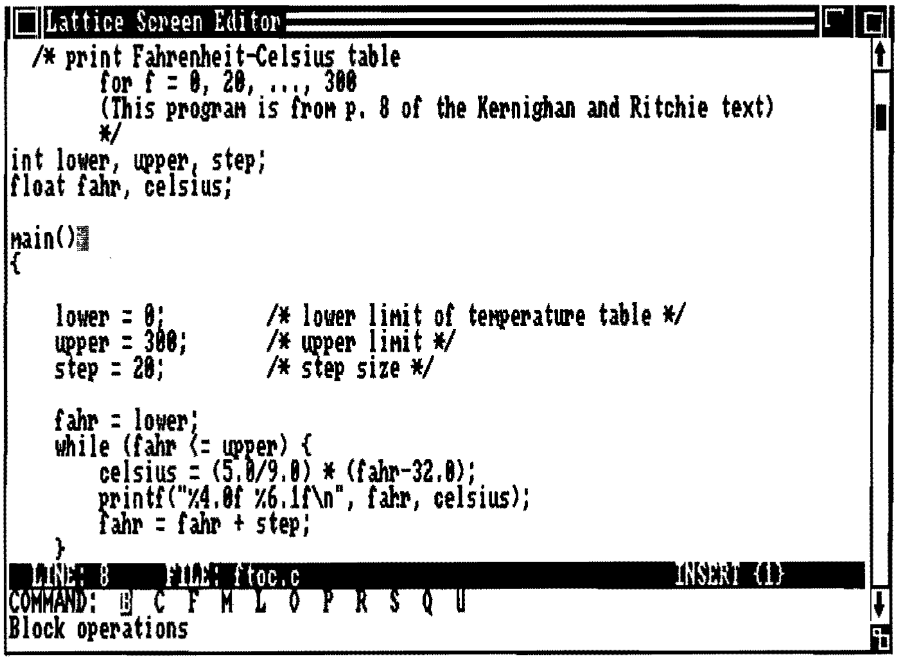


Figure 4.2 The Main Menu Display

To select a command from the *Main Menu*:

- Use the Amiga mouse to pull-down and click on the appropriate command, or
- Position the highlighted bar over the selection using the ← and → keys, then select the command by pressing the ↵, or
- Enter the first character of the command.

Pressing F2 while in the *Main Menu* or within any command sub-menu cancels the action and returns you to your text window.

4.12.2 Display Next Error (F5)

Use this key to search for errors within the compiler output after compiling a source file. If you have not activated error checking and you press F5, the message:

```
ERROR: LC ERROR CHECKING NOT ACTIVE
```

is displayed on the *status line*.

4.12.3 Enter Escape Character (Ctrl + Esc)

The Enter Escape Character function causes the next character input to **LSE** to be placed into the text without any processing. This allows control characters and non-ASCII characters to be placed into the text.

If the character following Ctrl + Esc is the character \, the next special key that you press places a coded string into your text representing that special key. The special key is not executed. This coded string representing the special key action is used by the keyboard macro processor when a macro file is loaded and replayed.

4.12.4 Find Next Match (Alt + S)

The Alt + S key initiates the continuation of the last search or search and replace. Alt + S continues the search or search and replace either forward or backward through the text file in the direction it was first specified. If there was no previous search **LSE** displays the message:

```
ERROR: NO PREVIOUS SEARCH
```

on the *status line*. If this is either the last occurrence of a specified string or if the string was not located in the file, the message:

```
text  NO MORE MATCHES FOUND
```

is displayed on the *status line* where *text* is the text string included within the Search or Replace.

Separate Search and Replace buffers are kept for each open window. If you changed windows since the last search and have not entered a new search string, Alt + S uses the search buffer of the non-active window.

The Search and Replace buffer is flushed for a window when performing:

- *Project Menu/Quit Option*
- *Project Menu/Save Option*

The Search and Replace Buffer is not flushed for a window when performing:

- *Project Menu/Open Option*
- *Project Menu/Next Option*
- *Project Menu/Flush Option*

4.12.5 Interlace Toggle (Ctrl + ENTER)

The Ctrl + ENTER key toggles the screen between either the default 20 lines to 45 lines per window.

4.12.6 Set Compiler Options (Ctrl + F4)

The Ctrl + F4 key is used to set the compiler options to be used when compiling C source files from memory. Pressing Ctrl + F4 causes the message:

```
ENTER OPTIONS FOR C COMPILER (for example, -S -d -i/lc/)
```

to be displayed on the *descriptor line* and the prompt:

```
LC_OPT:
```

on the *status line*. Enter the options for the Lattice C Compiler on the *input line*. Terminate the options by pressing ↵. Refer to your *User's Reference Manual* for complete information on Lattice C compiler options.

Section 5

Compiling From LSE

5.1 Introduction

Section 5 describes how to compile from **LSE** as an integrated edit/compile environment.

You can use **LSE** to both write your C source code and then compile your code directly from memory. Pressing F4 invokes the Lattice C Compiler using your predefined compiler and error checking options.

5.2 Integrated Environment

Using this integrated compile/edit environment you can compile the source file while buffering compiler syntax errors for later review. Prompts are displayed only during the review cycle. You can elect to check the compiler output either for errors only or for both errors and warnings.

5.3 Before Compiling

Before compiling your source file you can:

- Add options to the *LC* command using the Ctrl + F4 key combination.
- Determine conditions for handling syntax errors and warnings in the compiler output from the Mode Menu. The default error handling conditions are "prompt for each syntax error located" and "check compiler output for errors and warnings".

5.4 Compiling Your Source File

Once you press F4 your source file remains on the screen while it is compiled from memory and *Compiling...* is displayed on the *input line*.

When compiling your source files from memory, a series of intermediate messages are displayed describing each step of the process.

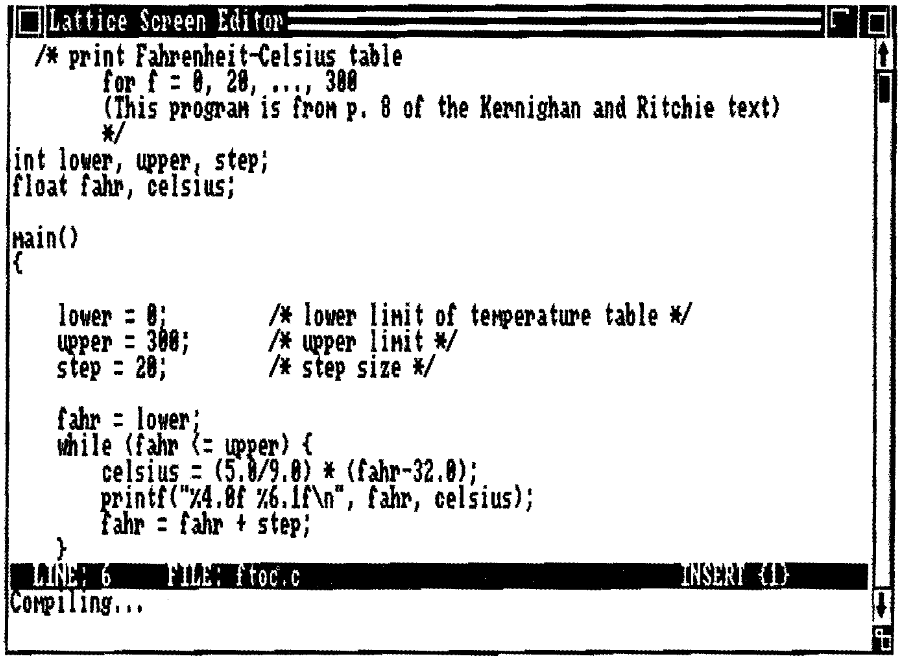


Figure 5.1 Invoking the Lattice C Compiler

The intermediate messages are displayed on the *descriptor line* and describe the current compiler activities. You are returned to your source file once a successful compile has been performed:

Refer to your *Compiler Command Reference* for information on the Lattice C compiling process.

If you try to compile a text file, that is if you press F4 when your current file has an extension other than *.c* the message:

```
ERROR: TEXT IS NOT .C FILE
```

is displayed on the *status line* and no action occurs. If your source code "blows up" during the compile, the message:

```
Fatal Compiler Error. Press any key to continue.
```

is displayed on the *status line* and no further action occurs. Press any key to leave the compiler and return to your source file.

5.5 Encountering Syntax Errors

When a syntax error is located in your source file it will be saved in a buffer. You can now review and correct your errors in the source file before you recompile. Figure 5.2 shows how the display looks when an errir is encountered.

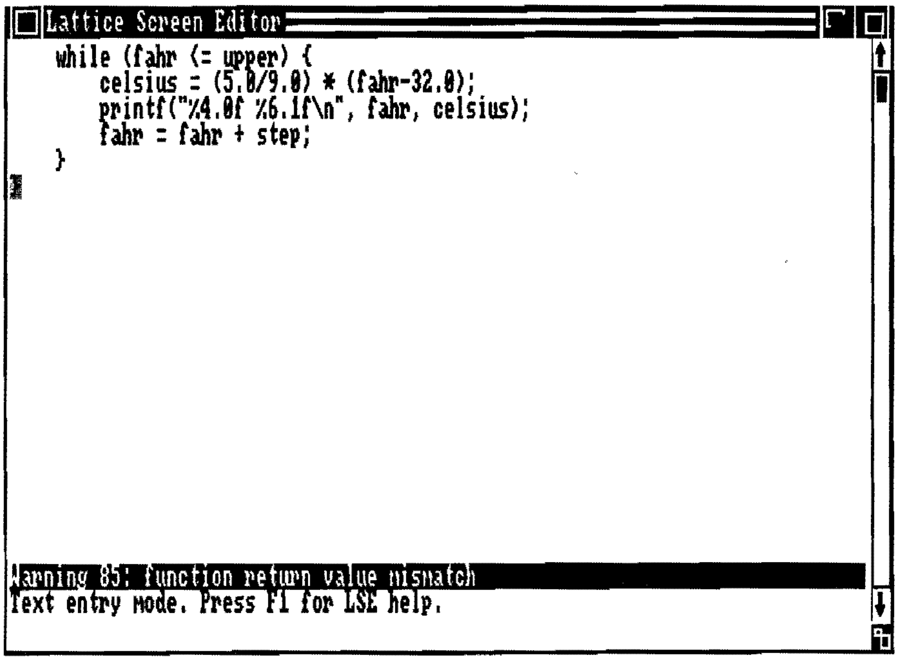


Figure 5.2 Compiler Warning Error

5.6 Reviewing Errors in Your Source File

You can review the errors in your source file once the source file has been compiled.

- In order to review your syntax errors you must have previously selected to save the errors rather than abort after the first error.
- You can also review your warnings, again if you previously chose to have the compiler check for warnings.

Error handling is an option on the **LSE** *Mode Menu* and can be changed for each editing session. See *Section 6* for complete information on the Mode menu. Your default error handling can be set using **LSEINST**. See *Appendix B* for complete information on **LSEINST**.

Errors are reviewed using [F5]:

- Press [F5] and you are positioned at the first line in your source file containing a syntax error.
- Press [F5] again to locate the next line containing a syntax error.
- Once all errors have been located, the message *NO MORE LC ERRORS* is displayed.

The errors can be located only for the source file compiled from the current window. Press [F5] while in a window other than that of your compiled source file and the message:

```
ERROR: LC ERROR CHECKING NOT ACTIVE IN CURRENT WINDOW
```

is displayed on the *status line*.

If you press [F5] when either no source file has been compiled or the source file has been compiled with no errors, the message:

```
ERROR: LC ERROR CHECKING NOT ACTIVE
```

is displayed on the status line and no other action is performed.

Section 6

The LSE Mode Menu

6.1 Introduction

Section 6 describes the **LSE** *Mode Menu*. The *Mode Menu* is displayed by pressing Ctrl+M or by selecting *Select LSE Mode* from the *Main Menu*. The *Mode Menu* controls the following **LSE** options:

- Input mode is to be used.
- Tab stops are to be used.
- Expand tabs into spaces.
- Automatic indenting is to occur on an open brace ({).
- Column Display is to be used.
- Patterns or simple string matching is to be used for searches.
- Prompt before executing an undo.
- Compiler output to be checked for warnings and errors or errors only.

6.2 Selecting A Mode Menu Option

Use the ↑ and ↓ keys to move up and down the list. Press the Space key to change a selection or to rotate through the choices for an item. Press the ↵ to save any changes. Press F2 to leave the *Mode Menu* without making any selections.

Each of the mode options has a default value. You can change this default value using **LSEINST** to tailor **LSE** to meet your needs and so that the mode option need not be changed for each editing session.

Some mode options affect all windows while others affect only the current window. To change an option for a window you must display the *Mode Menu* and select the specific option while in that window.

6.3 Input Modes Option

The Input Modes option controls the mode in which text is entered into **LSE**. There are five input modes to choose from:

- A default mode during which there is no post-processing of input.
- Three assembly language modes to perform case folding and special character field termination.
- Power-type (word-wrap) mode in which carriage returns are automatically inserted into the text when the screen size width is exceeded by the current line length.

The default Input Mode causes no post-processing of the input. The input mode affects only the current window.

6.3.1 Assembly Language Modes

The assembly language modes provide for the folding of character input into uppercase for one, two or three fields. After you have selected your case folding requirements, you are asked to specify the character that initiates a comment for your assembler.

In any of the assembly language modes:

- A colon (:), space (Space) or tab (→|) moves the cursor location from the first field into the second field.
- From the second field, a tab (→|) or a space (Space) moves to the third field, while the comment character moves to the comment field.
- From the third field, the comment character moves to the comment field.

There is never any case folding after the comment character has been entered. If the comment character is placed in the first field, the entire line is designated as a comment and no case folding occurs.

> **NOTE:** A comment character may be entered into the text without causing a move to the comment field by preceding the comment character with an escape character (\). Entering the comment character in this way preserves any active case folding.

6.3.2 Power-Type Mode

Power-type mode can be used for non-programming tasks, such as preparing documents or editing batch files. When the characters you are entering extend past the right margin, the entire last word is moved to the next line and processing continues. Note that no justification is performed when in power-type mode, resulting in a ragged right margin.

6.4 Tab Stops Option

The default tab stops within **LSE** are eight characters wide. The Tab Stops option is used to change to a different set of tab stop locations or to restore the settings back to the default settings. Changes to the tab stops affect only the current window.

You will be prompted to enter the new settings after indicating that you want them changed:

```
ENTER NEW TAB SETTINGS, SEPARATED BY COMMAS
```

Enter the new tab columns as a string of numbers with comma delimiters. The following string:

```
5,10,15,20,25 ↵
```

would create new tab stops five (rather than eight) columns apart.

6.5 Expand Tabs to Spaces Option

The default setting is to not expand tabs to spaces. Once **ON**, this option affects only the tabs entered while the option is **ON**. Tabs entered prior to this option being turned **ON** are treated as tabs. Once **OFF**, tabs entered while the option was on remain as spaces. The "expand tabs to spaces" option affects only the current window.

6.6 Auto Indenting Option

This feature can help structure C source code by providing automatic indenting on open and closed braces ({ }).

The default mode for Auto Indenting is **ON**. When **ON**, special cursor positioning occurs on a carriage return based upon the first non-blank character of the previous line:

- If the character is an open brace ({), the new line is indented one tab stop to the right of the previous line.
- If the character is a close brace (}), the new line is indented one tab stop to the left of the previous line.
- If any other character is in the first non-blank position, the new line is indented to the same position as the previous line.

When **OFF**, no automatic indenting occurs. This option affects the current window only.

6.7 Column Display Indicator Option

The Column Display Indicator can be displayed on the *status line*. This option toggles the Column Display Indicator **ON** and **OFF**. This option affects all windows. The default position is **OFF**.

6.8 Search Parameters Option

This option can toggle the **LSE** Search Parameters between Pattern Searches (the default) and Simple String Matching. The Search Parameters are used by both the *Search Command* and the *Replace Command*. This option affects all windows. The default is Pattern Searches.

6.9 Prompt Before Undo Option

This option can eliminate the prompt before an "Undo" when the *Undo Command* is selected. When the Undo prompt is displayed you can choose whether or not to continue the action. When this prompt is off, an Undo is performed automatically when the *Undo Command* is selected.

This option affects all windows. The default is to display the prompt before an Undo.

6.10 Check Compiler Output for Errors Option

The Check Compiler Output for Errors determines the level of error checking performed by the Lattice C Compiler. You can choose to:

- Check the compiler output for Errors and Warnings.
- Check the compiler output for Errors only.

This option affects all windows. The default is to check the compiler output for Errors and Warnings.

Appendix A

LSE Error Messages

A.1 Introduction

Appendix A presents a listing of the error messages which can be generated by **LSE** during start up and normal execution.

A.2 Start Up Errors

You cannot run **LSE** if you receive a start-up error message. If you can correct the problem, do so and start **LSE** again.

DATA FILE LSE.DAT NOT FOUND

The **LSE** data file, **LSE.DAT**, was not found either in the current directory or in any of the directories in the path when starting **LSE**. Add **LSE.DAT** to a valid directory and re-invoke **LSE**.

ILLEGAL OPTION

A command line option was entered that **LSE** did not recognize. The command line options must be entered correctly in order to invoke **LSE**.

MESSAGE FILE LSE.MSG NOT FOUND

The **LSE** message file, **LSE.MSG**, was not found either in the current directory or in any of the directories on the path. Add **LSE.MSG** to the proper directory and re-invoke **LSE**.

NOT ENOUGH MEMORY TO START LSE

There is not enough memory available within your computer for **LSE** to load **LSE.MSG** and **LSE.DAT**.

A.3 Processing Errors

Processing errors are displayed during the normal course of running **LSE**. Processing error messages are displayed on the *status line* of the current window. They are preceded by:

```
ERROR:
```

The error message is displayed on the *status line* until the cursor is moved off the current line.

BAD LINE NUMBER

The entry for a line number was answered with non-numeric input.

BAD SEARCH PATTERN

The input for a Search or Replace is not a valid pattern.

BLOCK NOT PROPERLY MARKED

A block option was requested without a properly marked text block.

CAN'T ASSIGN WITHIN AN ASSIGN

An attempt was made to turn on a macro keysaver while a macro keysaver was already active.

CAN'T EXECUTE MACROS DURING ASSIGN

An attempt was made to execute a macro while a macro keysaver was turned on.

CAN'T MOVE BLOCK FROM WITHIN BLOCK

An attempt was made to move a block to a location inside the block.

COMPILER REPORTED ERRORS

A source file compiled from memory with the Lattice C Compiler contained errors. The source file did compile through to end-of-file. The errors (if saved) can be reviewed using F5.

COMPILER ERROR BUFFER OVERFLOW

A source file compiled from memory with the Lattice C Compiler generated more errors than could be held in the error buffer. The source file is compiled through to end-of-file. The error buffer is not cleared. Review the errors contained in the error buffer using F5 and recompile the file.

ERROR OPENING OUTPUT FILE

An error occurred while **LSE** was trying to open a file for output. This generally indicates that you have tried to write to a read-only file or have run out of directory space.

ERROR READING INPUT FILE

An error occurred while **LSE** was trying to read information from a user file.

ERROR WRITING OUTPUT FILE

An error occurred while **LSE** was trying to write to an output file. This generally indicates that you have run out of disk space or that another program has the file open.

EXACT LINE NUMBER NOT FOUND

A line number was specified that does not exist in the file.

FATAL COMPILER ERROR

A source file compiled from memory with the Lattice C Compiler could not compile to completion due to a fatal error. You are returned to your source file. The compiler error message describing the fatal error is displayed on the *status line*.

FILE IS VIEW ONLY

An attempt was made to change or save a file opened in View Mode.

HELP NOT AVAILABLE

A request for context sensitive help was made while the help file **LSE.HLP** was not found at start-up time. There is no context sensitive help available for this editing session.

HELP NOT FOUND

LSE.HLP has been changed in such a way that the requested help message no longer exists. The index portion of the file has been changed or deleted.

HELP NOT FOUND IN HELP FILE

LSE.HLP has been changed in such a way that the requested help message no longer exists.

INSUFFICIENT MEMORY

LSE has run out of available memory while trying to add text to the file. As much of this text file as memory will allow will be saved; the remainder will be lost.

INSUFFICIENT MEMORY FOR HELP

The correct help message was found in **LSE.HLP** but there is not enough memory to create and display the help window.

LC ERROR CHECKING NOT AVAILABLE

An attempt was made to review compiler errors for the file in the current window while this file has either not been compiled or is not a C source file.

LC ERROR CHECKING NOT AVAILABLE IN CURRENT WINDOW

An attempt was made to review compiler errors for the file in the current window. The other window being displayed has a source file which was compiled. Make that window the current window and review the compiler errors.

MACRO FILE NOT FOUND

The specified macro file was not found either in the current directory or in a directory along the user's path.

MACRO TOO BIG, – MACRO ENDED

A macro was created having more than 255 keystrokes. The macro is ended and the first 255 keystrokes can be executed as a macro.

MAX LINESIZE EXCEEDED

More than 256 characters were entered on the current line.

NO ALTERNATE WINDOW AVAILABLE

An attempt was made to change to a second window while no second window was displayed.

NO AVAILABLE WINDOW FOR NEW FILE

An attempt was made to open another window when a maximum number of windows are already open.

NO FILE NAME SPECIFIED

An attempt was made to save a file which was started or opened without a file name. Use the *Project Menu/Rename Option* to name the file and save the file.

NO MORE MATCHES FOUND

The Find Next function was not able to locate the displayed search string.

NO PREVIOUS SEARCH

The Find Next function was used while no search string had been specified.

ONE WINDOW OPEN - DISPLAY SIZE CANNOT BE CHANGED

An attempt was made to change the display size of the screen area while only one window was open. The Window Size toggle is available only when two windows are open.

PRINTER ERROR

An attempt was made to print a block when the default printer was not available.

TEXT IS NOT .C FILE

An attempt was made to compile the current file from memory. The file is not a C source file (it does not have a file extension of .C) and cannot be compiled.

UNABLE TO LOAD COMPILER

An attempt was made to compile a source file from memory. The Lattice C Compiler either cannot be found using the current path or there is not enough memory to load the compiler once it was found. Check your path statement and try the compile again.

USER ABORTED SEARCH

A Search or Replace was aborted before its normal termination.

Appendix B

Customizing LSE

B.1 Introduction

Appendix B describes installing and customizing **LSE**. It covers the purpose and function of **LSEINST**, the installation program for **LSE**. It also describes tailoring **LSE** to your needs by making changes to the Message (**LSE.MSG**) and Help (**LSE.HLP**) Files.

B.2 LSEINST: The Installation Program

LSEINST is a screen-oriented program used to re-define the default keystrokes and options in **LSE**.

All keystrokes in **LSE** can be changed. The keystrokes are presented in the following groups:

- Cursor Movement Keys.
- Delete and Insert Keys.
- Command Menu Keys.
- Change Window Keys.

- Compiler Keys.
- Special Use Keys.
- Execute and Assign Macro Keys.

The **LSE** mode options can also be changed. The options identify:

- What input mode is to be used.
- What tab stops are to be used.
- Whether tabs are to be expanded to spaces.
- Whether automatic indenting is to occur on an open brace ({).
- Whether the Column Display is to be used.
- Whether patterns or simple string matching is to be used for searches.
- Whether a prompt is displayed before executing an undo.
- Whether the compiler output is checked for warnings and errors or errors only.

Each set of keystrokes and modes is presented on a single screen display which is selected from the **LSEINST** Main Menu.

B.2.1 The LSE Data File (LSE.DAT)

LSEINST stores all key and option information in the **LSE** Data File (**LSE.DAT**). When started, **LSEINST** attempts to read in your current **LSE.DAT** file.

- If **LSEINST** cannot find **LSE.DAT** in your current directory or along the directories in your path, it assumes you want to create a new **LSE.DAT**.
- If **LSEINST** does find your current **LSE.DAT** it assumes that you want the information from this file to be used as the defaults for this execution of **LSEINST**.

Upon leaving **LSEINST** you have the option to save or abandon your changes for this session with **LSEINST**.

B.2.2 Running LSEINST

LSEINST is started by entering:

```
LSEINST ↵
```

This causes the **LSEINST** main menu to be displayed.

The **LSEINST** main menu controls the group of items you want to change or view. Select the group by pressing the function key (F1 through F10) displayed to the left of the description. For example, press F4 to display the Command Menu keys display.

Use the Esc to return to this display from any other display within **LSEINST**.

B.2.3 Changing LSE Keystrokes

To change the default keystrokes:

- Select the desired display from the Main Menu.
- Press the function key (F1 through F8) appearing to the left of the description to display the key group.
- Once the selected key group is displayed, press the function key to the left of the description to reassign the key value for the identified item.
- After making the desired change(s), press Esc to return to the Main Menu.

To change the current definition of the Quit key from F3 to Ctrl + Q:

- Press F6.
- The message *Press key to be used for <GO TO QUIT LSE COMMAND>* is displayed.
- Press and hold Ctrl and then press Q.
- The current definition of the Quit key is then displayed as Ctrl + Q.

LSEINST identifies key conflicts but allows the key to be selected. For example, you wanted to change the Quit keystroke from **Q** to **L** (for *Leave*),

LSEINST would display the message:

```
<GO TO QUIT LSE COMMAND> conflicts with <GO TO LOCATE LINE
COMMAND>
```

You should avoid conflicts with the assigned **LSE** keys.

B.2.4 Changing LSE Options

As just stated above, the options identify the default processing performed by **LSE**. The defaults can be changed for each editing session using the *Mode Menu*. Select the options display by pressing F9 while the **LSEINST** main menu is displayed.

To change an option:

- Position the cursor at the option to be changed using the ↑ and ↓ keys.
- Change the current selection by pressing Space. If there is more than one selection, cycle through the choices by pressing Space.
- Select the setting by pressing ↵.
- Once you have completed your selections, press Esc to return to the main menu.

Refer to *Section 6* for information on the various mode options.

B.2.4.1 Default Options

LSE is shipped with the following default options:

- No processing of input.
- Set tab stops to default settings (every 8 spaces).
- Automatic indenting on '{' ON.

- Status line column display OFF.
- Searches use patterns.
- Prompt before performing undo.
- No default file extension.
- Rename current file with backup *.BAK* extension.
- Check compiler output for errors and warnings.

These options can be changed for an editing session by using the *Mode Command*. Use **LSEINST** to change the defaults.

B.2.4.2 File Extension Options

A file extension can automatically be added to a filename when specified without an extension. There are two file extension options:

- No default file extensions.
- Use default file extensions.

The "no default file extensions" is the system default. When you select default file extensions, the message and prompt:

```
CURRENT SETTING:
ENTER DEFAULT EXTENSION FOR FILENAMES (examples, C,ASM)
```

are displayed. Enter the extension you want to use as the default and press ⏎. It will then appear as the *CURRENT SETTING*. For example, if you make *c* your default extension, the file *ftoc* would be renamed to *ftoc.c* when edited.

The file extension option can be changed only within **LSEINST**.

B.2.4.3 Backup File Processing

You can choose to automatically generate backup copies of your files as they are saved either in a separate directory or with the specified file extension

within the same directory as your source file. There are three backup file processing options:

- Rename current file with backup extension.
- Do not create backup file.
- Place backup file in backup directory.

Renaming the current file with backup extension is the system default and *.bak* is the system default backup extension.

When you select backup directory, the message and prompt:

```
CURRENT SETTING:
ENTER DEFAULT BACKUP DIRECTORY (for example, C:/BAK)
```

are displayed. Enter the directory you want to use as the default and press ↵. It will then appear as the *CURRENT SETTING*. For example, if you make *C:/BAK* your default backup directory, the file *ftoc.c* would be saved to both the current directory and *C:/BAK/FTOC.C*.

When you select **rename current file with backup extension**, the message and prompt:

```
CURRENT SETTING:
ENTER DEFAULT BACKUP EXTENSION (for example, BAK)
```

are displayed. Enter the extension you want to use as the default and press ↵. It will then appear as the *CURRENT SETTING*. For example, if you make *.old* your default extension, the file *ftoc.c* would be backed-up to *ftoc.old* when saved.

The backup file processing option can be changed only within **LSEINST**.

B.2.5 Leaving LSEINST

Once you have completed your session with **LSEINST** return to the main menu and press F10. The exit menu is then displayed. From this menu you can choose to:

- Print the current key selections.
- Abandon the file and not save any changes made during this session.
- Rename the current data file.
- Save changes made during this session to the current data file.

Like all other displays, press the function key to the left of the description to perform the task described.

B.2.5.1 Print the Key Selections

Press F1 to print the current key selections to your default printer. Since this document only covers the default key selections, this option is very useful to track any changes you have made to the key assignments.

B.2.5.2 Abandon This Session

Press F2 to abandon this session of **LSEINST**. Any changes made to the **LSE** data file (**LSE.DAT**) are ignored.

B.2.5.3 Rename the Current Data File

Press F3 to rename the current data file. You may want to have a series of data files to control editing of specific types of files. This is accomplished by re-directing **LSE** to the named data file.

Once you press F3, the message and prompt:

```
CURRENT SETTING:  LSE.DAT
ENTER NEW DATA FILENAME
```

are displayed. Enter the new data file name and press ↵. The new name will then appear as the *CURRENT SETTING*. For example, if you enter *SOURCE.DAT* as your data file name it will appear as the current setting. After saving the changes, **LSE** would use *SOURCE.DAT* rather than *LSE.DAT* to locate key and option information.

B.2.5.4 Save the Changes Made This Session

Press F4 to save changes made during this session of **LSEINST**. After you press F4, the changes are saved to the current data file, the message:

```
INSTALLATION SUCCESSFULLY COMPLETED
```

is displayed.

B.3 Making Changes to LSE.MSG

LSE.MSG contains the text for all prompts, help messages and error messages used within **LSE**. It also contains information about default file extensions and coded information about menus. You can change any of the messages to alter the appearance and functionality of **LSE**.

Unlike **LSE.DAT**, there is no special utility for changing **LSE.MSG**; **LSE.MSG** is a text file containing over one hundred numbered message lines which can be changed using **LSE**.

B.3.1 Changing Menu Messages

Messages **1** through **8** contain the menus used during **LSE** processing. The menus are represented as a series of multi-line messages; each series defining an **LSE** menu. The messages have the following general layout:

```
#;"  @key{menu}{descriptor line}'
```

where:

#;	Is the message number.
@key	Is the default setting for this menu option and should never be changed.
{menu}	Is the string displayed on the *menu line* selected to initiate this command.

{descriptor line} Is the text that appears on the *descriptor line* when the menu item is pointed to.

The entire message string must be enclosed within quotation marks (" "). If there are several lines corresponding to the same message use the single quote character (') to indicate continuation and end the last line with the quote character (").

Each menu item should be given a default setting for the menu. This default setting identifies the position of the select bar when the menu is displayed. Use the character > to identify the default setting for the menu. It replaces the character @ directly preceding the *key*.

What follows is the default *Main Menu* from the Message File:

```
1;"COMMAND:   >B{B}{Block operations}'
1;"   @L{C}{Invoke Lattice C Compiler}'
1;"   @F{F}{Fork command processor}'
1;"   @M{M}{Select LSE mode}'
1;"   @N{L}{Go to a specific line number}'
1;"   @O{O}{Open new window}'
1;"   @C{P}{Project Menu}'
1;"   @R{R}{Search and replace}'
1;"   @S{S}{Search}'
1;"   @Q{Q}{Quit}'
1;"   @U{U}{Undo Last Change}"
```

Note that the *Block Command* is the menu default and that quotes are used to complete the menu selections following the *Undo Command*.

You may modify the text to be displayed on the *menu line*. This may, however, change the key you must press to choose the selection. For example, if you were to replace the *Block Command* with the following line:

```
1;"   @B{P}{Cut and Paste Operations}"
```

you would see *Cut and Paste Operations* on this menu and need to choose P to select this option.

You should not have conflicting choices on the option keys of the menu

selections since only the first option using that letter can be chosen. In the above example, *P* is selected rather than *C* as there would be a conflict with the *Invoke the Compiler Command.*

B.4 Making Changes to LSE.HLP

LSE.HLP contains the text that forms the help messages for the context sensitive help command of **LSE**. You may change these messages to make the help more suitable to your needs. Like **LSE.MSG**, **LSE.HLP** can be changed using **LSE**.

The text for the subjects are separated from each other by *subject numbers*. The *subject numbers* are two digits preceded by a period:

```
.00
```

In the sample page, the Block Read option is identified by *subject number .04* and the *Project Command* is identified by *subject number .06*. The text for the first help screen is identified by this *subject number*. The *subject number* controls what text goes with which element of **LSE** so these should not be changed.

The help text can span any number of pages. When multiple pages of help text are available, pressing any key will display the next page. Only that text appearing within the display window area can be displayed. There is no facility to display text past column 80.

Appendix C

Special Keys Summary

C.1 Introduction

Appendix C presents a summary of the keystrokes used within **LSE**. The summary covers the standard keystroke assignments which are delivered with the product and subsequently referred to within the text. The standard keystroke assignments can be customized to any control key or other special key using **LSEINST** as described in *Appendix B*.

C.2 Cursor Movement Keys

Function	Key	Action
Up One Line	↑	Move the cursor up one line.
Down One Line	↓	Move the cursor down one line.
Previous Character	←	Move the cursor one character left.
Previous Word	Ctrl + ←	Move the cursor one word left.

Next Character	→	Move the cursor one character right.
Next Word	Ctrl + →	Move the cursor one word right.
Beginning of Line	Home	Move the cursor to the first character of the current line.
Beginning of Text	Ctrl + Home	Move the cursor to the first character of text file.
End of Line	End	Move the cursor to the last character of the current line.
End of Text	Ctrl + End	Move the cursor to the last character of text file.
Previous Page	PgUp	Display the previous page of text.
Next Page	PgDn	Display the next page of text.
Scroll Up	Ctrl + ^	Scrolls the display up without moving the cursor.
Scroll Down	Ctrl + V	Scrolls the display down without moving the cursor.

C.3 Change Window Keys

Function	Key	Action
Change Window	Ctrl + F6	Change active window when in the split screen mode.
Change Window Size	F9	Toggle window display size from full to half screen.
Cycle Windows	F6	Cycle through hidden windows.

C.4 Insert Keys

Function	Key	Action
Insert Mode Toggle	Ins	Toggle between the Insert Mode and the Overwrite Mode.
Insert a Line	Ctrl + N	Insert a line before the current line, do not break the current line.
Restore Deleted Line	Alt + Y	Restore the last line deleted with Ctrl + Y at the cursor position, shifting the current line down one line.

C.5 Delete Keys

Function	Key	Action
Delete Current Character	Del	Delete current character from text.
Delete Previous Character	BkSp	Delete previous character from the text.
Delete Word	Ctrl + W	Delete text until the next word is encountered.
Delete Line	Ctrl + Y	Delete the current line. The line can be restored using Alt + Y.
Delete to End of Line	Ctrl + E	Delete text to the end of the current line.

C.6 Help Key

Function	Key	Action
Context Help	F1	Display help window.
Context Help	HELP	Display help window.

C.7 Change Color Keys

Function	Key	Action
Cycle Foreground	F7	Cycle through the foreground colors.
Cycle Background	F8	Cycle through the background colors.

C.8 Keysaver Macro Keys

Function	Key	Action
Start/Stop Keysaver Macro	Alt + #	Start or stop recording keys to the keysaver macro.
Replay a Macro	Alt + F#	Replay a macro.

C.9 Mark Block Keys

Function	Key	Action
Mark Beginning of Block	Ctrl + [	Mark the start of a block of text.
Mark End of Block	Ctrl +]	Mark the end of a block of text.

C.10 Command Menu Key Functions

Function	Menu	Direct	Description
Block	B	Ctrl + B	Block Operations
Compile	C	F4	Invoke Lattice C Compiler
Fork	F	Ctrl + F	Fork command processor
Mode	M	Ctrl + M	Select LSE Mode
Line Number	L	Ctrl + L	Go to a specific line number
Open	O	Ctrl + O	Open New Window
Project	P	Ctrl + P	Project Menu
Replace	R	Ctrl + R	Search and replace
Search	S	Ctrl + S	Search
Quit	Q	F3	Quit
Undo	U	Ctrl + U	Undo Last Change

C.11 Action Keys

Function	Key	Action
Menu Toggle	F2	Display *Main Menu* or cancel action and return to editing.
Display Next Error	F5	Frame next compiler error.
Escape Character	Ctrl + Esc	Enter escape character.
Find Next Match	Alt + S	Locate next match for search.
Interlace Toggle	Ctrl + ENTER	Interlace Toggle key.
Set Compiler Options	Ctrl + F4	Set compiler options.

Appendix D

Regular Expression Syntax

D.1 Introduction

Appendix D describes the regular expression syntax and minor deviations that the **LSE** version makes from the standard. The following text is taken from the *Lattice Compiler Companion Users Guide* for the **Grep** utility. The functions that perform the regular expression search in **LSE** are the same as those provided with this package.

D.2 Pattern Search

For the purpose of this appendix, a regular expression will be referred to as a pattern.

The version of a regular expression search used within **LSE** is very much like the regular expression search of UNIX grep, but with a few exceptions. A slightly less general class of patterns (regular expressions) is supported; in particular, the UNIX regular expressions of the form:

```
\{m\}  or  \{m,\}  or  \{m,n\}
```

which match ranges of occurrences of regular expressions, are not recognized by the regular expression search in LSE. Neither are the regular expressions of the forms:

```
\(...\)   or))\n
```

where **n** is a numeral, and **\n** is intended to match the same string of characters as was matched by the expression enclosed between \(and \).

D.3 Similar to UNIX grep

Aside from these departures, this regular expression search has all the features of the UNIX grep. In particular:

- An ordinary character is a one-character pattern that matches itself.
- A backslash (\) followed by any special character is a one-character pattern that matches the special character itself, *except* when it is used in an escape sequence for a non-graphic character. The special characters include:

  . * [and **** . (Unlike UNIX, **. *** and **** remain special even within square brackets, but **[** does not.)

  ^ – is special at the *beginning* of an entire pattern.

  $ – is special at the *end* of an entire pattern (as explained below).

  ! – is special immediately following a **[** (as explained below).
- A period (.) is a one-character pattern that matches any character except newline or end-of-string.
- A non-empty string of characters enclosed in square brackets ([ ]) is a one-character pattern that matches *any one* character in that string. However, if the first character of the string is **!** then the one-character pattern matches any character *except* newline end-of-string, and the remaining characters in the string. Thus ! is a *not* sign applied to a character class. Note that this is a notational departure from UNIX, which uses ^ for this application (thus rendering ^ ambiguous).

Also supported are such character classes as:

```
[a-z]
```

indicating all of the letters **a** through **z**. The only restriction in such cases is that the first character must occur alphabetically prior to the second.

- **LSE** regular expression search recognizes certain non-graphic characters denoted in the manner of Kernighan & Ritchie. In particular, the expression on the left below may be used to denote a pattern that is matched by the character described on the right:

\n	newline
\t	tab
\b	backspace
\r	carriage return
****	backslash
\xmn	the character whose hex number is denoted by the hex numerals **m** and **n** (in sequence)

- Text strings containing whitespace (spaces and tabs) may be specified as as a regular expression search pattern by enclosing them in quote marks (" "). Thus a string may be searched for by means of:

```
"this is a string"
```

If **r** is a pattern, then:

```
r*
```

matches *one or more* occurrences of **r**. If there is a choice, then the longest leftmost string that matches is chosen.

If **r** is a pattern, then:

```
r+
```

matches *one or more* occurrences of `r`. If there is a choice, then the longest leftmost string that matches is chosen.

The concatenation of patterns is a pattern that matches the concatenation of the strings matched by each component of the pattern.

Appendix E

LSE AREXX Interface

E.1 Introduction

Appendix E describes the AREXX support in LSE. It covers the basic commands supported and how to access them. It also shows an example of utilizing this powerful interface.

E.2 AREXX Command summary

All of the baisc LSE functions currently supported by **LSEINST** are available as simple two letter commands. They are summarized as follows:

Command	Key Description	Current Definition
PC	Previous Character	NUM 4
NC	Next Character	NUM 6
PL	Previous Line	NUM 8
NL	Next Line	NUM 2
PP	Previous Page	NUM 9
NP	Next Page	NUM 3

Command	Key Description	Current Definition
BT	Beginning Text	Ctl-NUM 7
ET	End Text	Ctl-NUM 1
DC	Delete Character	NUM .
DE	Delete to End of Line	Ctl-e
DL	Delete Line	Ctl-y
IL	Insert Line	Ctl-n
BL	Beginning of Line	NUM 7
EL	End of Line	NUM 1
SD	Scroll Down	Ctl-v
SU	Scroll up	Ctl-6
MB	Mark Beginning of Block	Ctl-[
ME	Mark End of Block	Ctl-]
PW	Previous Word	Ctl-NUM 4
NW	Next Word	Ctl-NUM 6
DW	Delete Word	Ctl-w
CW	Change Windows	NUM 5
SA	Go To Next String Match	Alt-s
ES	LSE Escape Character	Ctl-
IN	Insert Mode Toggle	NUM 0
CO	Set C Compiler Options	Ctl-F4
NW	Cycle through Open Windows	F6
MM	Go to Main Lse Menu	F2
HE	Get LSE Help	F1
FB	Toggle Window Front to Back	F10
OM	Go to Mode Menu	Ctl-m
NE	Go to next Compiler Error	F5
LC	Invoke Lattice C Compiler	F4
CS	Change Display Size	F9

Command	Key Description	Current Definition
BM	Go to Block Menu	Ctl-b
PM	Go to Project Menu	Ctl-p
FD	Go to Fork Dos Command	Ctl-f
RK	Restore Kill Buffer	Alt-y
GL	Go to Locate Line Command	Ctl-l
OW	Go to Open Window Command	Ctl-o
SR	Go to Search/Replace Command	Ctl-r
SE	Go to Search Command	Ctl-s
QU	Go to Quit LSE Command	F3
UN	Go to Undo Command	Ctl-u
M1	Execute Macro 1	Alt-F1
M2	Execute Macro 2	Alt-F2
M3	Execute Macro 3	Alt-F3
M4	Execute Macro 4	Alt-F4
M5	Execute Macro 5	Alt-F5
M6	Execute ARexx Macro 1	Alt-F6
M7	Execute ARexx Macro 2	Alt-F7
M8	Execute ARexx Macro 3	Alt-F8
M9	Execute ARexx Macro 4	Alt-F9
M0	Execute ARexx Macro 5	Alt-F10
A1	Assign Macro 1	Alt-1
A2	Assign Macro 2	Alt-2
A3	Assign Macro 3	Alt-3
A4	Assign Macro 4	Alt-4
A5	Assign Macro 5	Alt-5
IM	Toggle Interlace Mode	ENTER

With the AREXX interface, there is additional functionality needed to fully access the environment. For this, we have added some additional commands that have no keystroke equivalent.

COMMAND	DESCRIPTION	EXAMPLE
UC	Use the following charater(s)	'UChello'
CR	Carriage control	'CR'
GC	Get column of cursor	'GC'
GI	Get index of cursor	'GI'
GL	Get number of current line	'GL'
DM	Display message	'DMExample Message'
GE	Get error from last command	'GE'
GT	Get text of current line	'GT'
GM	Prompt user for input	'GM'

Commands may be strung together seperated by spaces. However, if the UC command is encountered, the rest of the line is considered to be text.

E.3 A complex AREXX Example

The addition of a **AREXX** interface makes **LSE** a much more powerful editor. Listed below is a sample **AREXX** macro that is useful for programs that often renumber a list of **#define** statements. It will take a marked block, and renumber the last token on each line sequentially. For example

```
#define foo1  1
#define foo2  1
#define foo3  1
#define foo4  1
```

will be changed to

```
#define foo1  1
#define foo2  2
#define foo3  3
#define foo4  4
```

This is accomplished with the following **AREXX** macro:

```
/* rexx macro to renumber #define statements */
/* Get the line number of the end of the block */
'BM UCE'
options results
'GL'
end = result
options

/* Go to the start of the block */
'BM UCB'
options results

/* Get the line at start of block */
'GT'
options
line = result

/* Go to the last word on the line */
'EL PW'
options results

/* get the index of the last work */
'GI'
options
index = result

/* convert the last word to a number */
num = SUBSTR(line, index+1, length(line) - index)

/* delete the last word of the next line */
'NL EL PW DE'
do forever

/* increment number */
num = num + 1

/* replace with the new number */
num

/* test if we're at the last line of the block */
options results
```

```
'GL'
options

/* exit if done with block */
if result >= end then exit(0)

/* delete last word of next line and repeat */
'NL EL PW DE'
end
```

Index

O

P

Q

R

S

T

U

V

W

Lattice

Lattice C Compiler For Amiga

User's Guide
Utilities
Commands
Editor

Volume 1